

**A PROJECT REPORT**

**on**

**“Transformative Text Summarization for Enhanced  
Information Retrieval”**

**Submitted to**

**KIIT Deemed to be University**

**In Partial Fulfillment of the Requirement for the Award of**

**BACHELOR’S DEGREE IN**

**Computer Science & System Engineering**

**BY**

Debayan Das	2028180
Sudipta Mali	2028185
Shivansh Awasthi	2028195
Rhythm Sahu	20051022

**UNDER THE GUIDANCE OF**

**Prof. Sumita Das**



**SCHOOL OF COMPUTER ENGINEERING**  
**KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY**  
**BHUBANESWAR, ODISHA - 751024**

**December 2024**

# KIIT Deemed to be University

School of Computer Engineering  
Bhubaneswar, ODISHA 751024



## CERTIFICATE

This is certify that the project entitled

“Transformative Text Summarization for Enhanced Information  
Retrieval“

submitted by

Debayan Das	2028180
Sudipta Mali	2028185
Shivansh Awasthi	2028195
Rhythm Sahu	20051022

is a record of bonafide work carried out by them, in the partial fulfillment of the requirement for the award of Degree of Bachelor of Engineering (Information Technology) at KIIT Deemed to be university, Bhubaneswar. This work is done during the year 2023-2024, under our guidance.

Date: 07/12/2023

Prof. Susmita Das  
Project Guide

# Acknowledgements

We are profoundly grateful to **Prof. Susmita Das** of **Computer Science & System Engineering** for his expert guidance and continuous encouragement throughout to see that this project meets its target since its commencement to its completion.

Sudipta Mali  
Shivansh Awasthi  
Debayan Das  
Rhythm Sahu

# ABSTRACT

With the help of state-of-the-art NLP techniques, this ground-breaking project, "Transformative Text Summarization for Enhanced Information Retrieval," aims to transform information processing.

Through the use of cutting-edge algorithms, such as deep learning and machine learning models, the project aims to condense large amounts of textual data into brief but insightful summaries. This report illustrates the effectiveness and versatility of the system with a variety of content types through thorough examination and analysis.

The results highlight the developed model's transformative potential and provide a significant advancement in NLP for more efficient and effective information retrieval processes.

**Keywords:** Transformative, Text Summarization, Information Retrieval, Natural Language Processing (NLP), Algorithms

# Contents

1	Introduction	
2	Basic Concepts/ Literature Review	
3	Problem Statement / Requirement Specifications	
4	Implementation	
5	Conclusion and Future Scope	
	5.1	Conclusion
	5.2	Future Scope
References		
Individual Contribution		
Plagiarism Report		

# Introduction:

Natural Language Processing (NLP) and information science have come together in a novel way with "Transformative Text Summarization for Enhanced Information Retrieval." By using state-of-the-art NLP techniques, the project tackles the problems caused by an abundance of information. Its uniqueness resides in its capacity to extract significant insights from large amounts of textual data.

With an emphasis on effective information retrieval, the project uses sophisticated algorithms and machine learning models to create succinct yet thorough summaries. This new method aims to improve the accuracy and speed of information retrieval, which will add a great deal to the conversation about how NLP and efficient content curation work together in the digital age.

The project's innovative approach to information retrieval methods not only advances the field of natural language processing (NLP) but also offers a strategic solution to the complex problems of managing information in our data-rich environment.

Through the introduction of novel methodologies, the project methodically addresses the current issues surrounding information overload. Using cutting-edge natural language processing (NLP) techniques, it emphasizes the extraction of relevant insights from large amounts of textual data.

One of the project's unique selling points is its ability to effectively distill complex information into brief but thorough summaries. It makes use of cutting-edge algorithms and complex machine learning models to improve the accuracy and speed of information retrieval procedures.

Beyond its immediate uses, this project makes a significant contribution to the larger conversation about the changing nature of the relationship between natural language processing (NLP) and the need for effective content curation in the quickly developing information age.

Thus, this revolutionary approach not only propels NLP forward but also offers a thoughtful and strategic resolution to the complex problems related to information management.

Just to showcase an example, here is the screenshot from the code:

Dialogue:

Eric: MACHINE!

Rob: That's so gr8!

Eric: I know! And shows how Americans see Russian ;)

Rob: And it's really funny!

Eric: I know! I especially like the train part!

Rob: Hahaha! No one talks to the machine like that!

Eric: Is this his only stand-up?

Rob: Idk. I'll check.

Eric: Sure.

Rob: Turns out no! There are some of his stand-ups on youtube.

Eric: Gr8! I'll watch them now!

Rob: Me too!

Eric: MACHINE!

Rob: MACHINE!

Eric: TTYL?

Rob: Sure :)

Summary:

Eric and Rob are going to watch a stand-up on youtube.



# Basic Concepts/ Literature Review:

The literature review covers the fundamental ideas of text summarization with a focus on natural language processing (NLP) techniques.

Semantic analysis techniques, machine learning models optimized for summarization tasks, and pivotal information extraction algorithms are at the core of this exploration. These elements constitute the foundation of knowledge in the area. A thorough analysis of current models not only reveals their advantages and disadvantages but also acts as an essential basis for the creation of a novel text summarization technique.

The review creates a strong foundation by combining these essential components, opening the door for the ideation and application of a fresh method for text summarization.

## **Text Summarization Algorithms:**

An overview of conventional text summarization algorithms, such as abstraction- and extraction-based techniques, is given in this section.

It charts the development of these methods, emphasizing current developments in the area.

A comparative study is conducted to evaluate the benefits and drawbacks of different algorithmic approaches, providing information about the evolution from traditional to more advanced tactics in the last few years.

## **Advancements in NLP:**

Discover the revolutionary effects of machine learning and deep learning applications as you examine important advances in natural language processing (NLP) related to text summarization.

---

Examine the array of prominent frameworks and libraries that are essential to NLP research, illuminating the technological foundation that propels improvements in text processing and summarization techniques.

## **Review of Existing Models:**

Leading text summarization models are evaluated critically to provide complex insights into their advantages and disadvantages.

This assessment highlights the shortcomings and restrictions of the existing methods, laying the groundwork for future innovation.

Finding opportunities for development becomes critical to increasing the usefulness and efficacy of text summarization techniques.

## **Semantic Analysis Methodologies:**

This section delves into text processing techniques related to semantic analysis, revealing the complex relationship between semantic analysis and effective summarization.

Examining the complex relationship, it takes into account how important semantic understanding is to improving content extraction.

This investigation, which clarifies the relationship between summarization and semantics, emphasizes the need of using semantic analysis techniques to accomplish more sophisticated and successful text extraction, setting the stage for future developments in the field.

## **Evolution of Summarization Techniques:**

Examining the text summarization's historical development reveals important turning points.

Important turning points indicate the development of summarization techniques from their inception to modern applications. These methods have been greatly impacted by technological developments, which have brought about revolutionary changes in the way information is distilled.

Modern technology has accelerated the development of summarization techniques, taking them from simple starting points to complex, multifaceted approaches that capture the dynamic relationship between history, technology, and continuous improvement of text summarization methods.

## **Influential Models and Methodologies:**

An extensive investigation of key models influencing the field is followed by a review of approaches that have set standards for text summarization.

The review carefully selects important studies and papers, offering a perceptive overview of the major contributions that have influenced the field of text summarization research.

This analysis not only highlights the field's evolution but also lays the groundwork for the creation of a novel strategy that expands on the groundwork established by these influential models and methodologies.

## **Problem Statement / Requirement Specifications:**

Information overload is caused by the growing amount of textual data, which presents a significant obstacle to efficient information retrieval.

Although useful, current text summarization techniques are not able to meet the demands of efficiently retrieving complex content in smaller chunks. By creating a transformative text summarization strategy using cutting-edge Natural Language Processing (NLP) techniques, this project seeks to close this gap.

The problem statement focuses on the requirement for a novel approach that, in addition to improving information retrieval speed and accuracy, advances the conversation about the benefits of combining natural language processing (NLP) with effective content curation in the face of an abundance of information.

The goal of Transformative Text Summarization for Enhanced Information Retrieval is to create a system that can quickly and contextually richly summarize sizable amounts of text.

Advanced natural language processing methods for understanding and extracting important information, deep learning models for context-aware summarization, and integration with cutting-edge information retrieval systems are among the prerequisites.

The system ought to facilitate the summarization of multiple documents, manage various kinds of content, and ensure that the summaries remain coherent and informative. Furthermore, user feedback systems and flexibility in response to changing linguistic subtleties are essential for ongoing development. Scalability, ensuring effective processing of large datasets, and compatibility with a variety of document formats are all important considerations for the implementation.

By finding a medium ground between summarization length and content relevance, the system seeks to greatly improve users' information retrieval efficiency in a variety of domains.

# Implementation

```
[ ] from transformers import pipeline, set_seed

import matplotlib.pyplot as plt
from datasets import load_dataset
import pandas as pd
from datasets import load_dataset, load_metric

from transformers import AutoModelForSeq2SeqLM, AutoTokenizer

import nltk
from nltk.tokenize import sent_tokenize

from tqdm import tqdm
import torch

nltk.download("punkt")

[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
True
```

The code begins by importing various Python libraries:

1. **transformers**: This library is used for working with pre-trained models from the Hugging Face Model Hub.
2. **pipeline**: It provides a simple API for using pre-trained models for various tasks.
3. **set\_seed**: It sets the random seed for reproducibility.
4. **matplotlib.pyplot**: A library for creating visualizations and plots.
5. **datasets**: Hugging Face's library for handling datasets.
6. **pandas**: A data manipulation and analysis library.
7. **AutoModelForSeq2SeqLM** and **AutoTokenizer**: Classes from transformers for automatically loading pre-trained sequence-to-sequence models and their tokenizers.
8. **NLTK**: Natural Language Toolkit, used for natural language processing tasks.
9. **TQDM**: A library for adding progress bars to loops.
10. **torch**: PyTorch library for deep learning.

```
def generate_batch_sized_chunks(list_of_elements, batch_size):
    """split the dataset into smaller batches that we can process simultaneously
    Yield successive batch-sized chunks from list_of_elements."""
    for i in range(0, len(list_of_elements), batch_size):
        yield list_of_elements[i : i + batch_size]
```

1. `generate_batch_sized_chunks`: This is the name of the function.
2. `list_of_elements`: It is the input list that you want to split into batches.
3. `batch_size`: It is the desired size of each batch.
4. `"""split the dataset into smaller batches that we can process simultaneously Yield successive batch-sized chunks from list_of_elements."""`: This is a docstring, which is a string that provides documentation for the function. It explains the purpose of the function.
5. `for i in range(0, len(list_of_elements), batch_size):`: This is a for loop that iterates over the indices of the input list in steps of `batch_size`.
6. `yield list_of_elements[i : i + batch_size]`: The `yield` keyword is used to turn the function into a generator. It generates batches of elements by slicing the input list from index `i` to `i + batch_size`. The `yield` statement returns the batch and suspends the function's state, allowing it to resume from where it left off when the next batch is requested.
7. This generator function allows you to process a large dataset in smaller batches, which can be useful for tasks like training machine learning models in mini-batch fashion. Instead of loading the entire dataset into memory, you can process one batch at a time, reducing memory requirements and potentially speeding up the processing, especially when dealing with large datasets.

8.

```
def calculate_metric_on_test_ds(dataset, metric, model, tokenizer,
                               batch_size=16, device=device,
                               column_text="article",
                               column_summary="highlights"):
    article_batches = list(generate_batch_sized_chunks(dataset[column_text], batch_size))
    target_batches = list(generate_batch_sized_chunks(dataset[column_summary], batch_size))

    for article_batch, target_batch in tqdm(
        zip(article_batches, target_batches), total=len(article_batches)):

        inputs = tokenizer(article_batch, max_length=1024, truncation=True,
                           padding="max_length", return_tensors="pt")

        summaries = model.generate(input_ids=inputs["input_ids"].to(device),
                                   attention_mask=inputs["attention_mask"].to(device),
                                   length_penalty=0.8, num_beams=8, max_length=128)
        ''' parameter for length penalty ensures that the model does not generate sequences that are too long. '''

        # Finally, we decode the generated texts,
        # replace the token, and add the decoded texts with the references to the metric.
        decoded_summaries = [tokenizer.decode(s, skip_special_tokens=True,
                                              clean_up_tokenization_spaces=True)
                              for s in summaries]

        decoded_summaries = [d.replace("'", " ") for d in decoded_summaries]

        metric.add_batch(predictions=decoded_summaries, references=target_batch)

    # Finally compute and return the ROUGE scores.
    score = metric.compute()
    return score
```

This function, `calculate_metric_on_test_ds`, is designed to calculate a given metric (such as ROUGE) on a test dataset using a pre-trained sequence-to-sequence model. Let's break down the code step by step:

```
def calculate_metric_on_test_ds(dataset, metric, model, tokenizer,
                                batch_size=16, device=device, column_text="article",
                                column_summary="highlights"):
```

1. `dataset`: The input dataset containing text and summary pairs.
2. `metric`: The metric (e.g., ROUGE) used for evaluation.
3. `model`: The pre-trained sequence-to-sequence model.
4. `tokenizer`: The tokenizer associated with the model.
5. `batch_size`: The size of batches to process data.
6. `device`: The device on which the model should run (default is device which is assumed to be defined earlier).
7. `column_text`: The column name for the article text in the dataset.
8. `column_summary`: The column name for the target summaries in the dataset.

```
article_batches = list(generate_batch_sized_chunks(dataset[column_text],
batch_size))
target_batches =
list(generate_batch_sized_chunks(dataset[column_summary], batch_size))
```

1. The `generate_batch_sized_chunks` function is used to split the article text and target summaries into batches of the specified size.
- 2.
3. for `article_batch, target_batch` in `tqdm( zip(article_batches, target_batches), total=len(article_batches))`:
4. The `tqdm` function is used to create a progress bar to track the processing of batches.

```
inputs = tokenizer(article_batch, max_length=1024, truncation=True,
padding="max_length", return_tensors="pt")
```

1. The article text is tokenized using the tokenizer, ensuring that the input does not exceed a maximum length of 1024 tokens. Padding is added to make all sequences in a batch have the same length.

```
summaries = model.generate(input_ids=inputs["input_ids"].to(device),
attention_mask=inputs["attention_mask"].to(device), length_penalty=0.8,
num_beams=8, max_length=128)
```

1. The model generates summaries based on the tokenized article text. Parameters like `length_penalty`, `num_beams`, and `max_length` control the generation process.

```
decoded_summaries = [tokenizer.decode(s, skip_special_tokens=True,
clean_up_tokenization_spaces=True) for s in summaries]
```



1. The generated summaries are decoded using the tokenizer, removing special tokens and cleaning up tokenization spaces.

```
decoded_summaries = [d.replace("", " ") for d in decoded_summaries]
```

1. The decoded summaries are further processed to replace empty strings with spaces.

```
metric.add_batch(predictions=decoded_summaries,  
references=target_batch)
```

1. The predictions (generated summaries) and references (target summaries) are added to the metric for batch-wise evaluation.

```
# Finally compute and return the ROUGE scores. score = metric.compute()  
return score
```

1. The final ROUGE scores are computed and returned.
2. Overall, this function tokenizes the input articles, generates summaries using a pre-trained model, decodes the generated summaries, and evaluates them against reference summaries using a specified metric. It processes the data in batches, making it memory-efficient for large datasets.

#### ▼ Load data

Link: <https://huggingface.co/datasets/samsum>

```
[ ] dataset_samsum = load_dataset("samsum")  
  
split_lengths = [len(dataset_samsum[split]) for split in dataset_samsum]  
  
print(f"Split lengths: {split_lengths}")  
print(f"Features: {dataset_samsum['train'].column_names}")  
print("\nDialogue:")  
  
print(dataset_samsum["test"][1]["dialogue"])  
  
print("\nSummary:")  
  
print(dataset_samsum["test"][1]["summary"])
```

1.

2. `dataset_samsum = load_dataset("samsum")`: Loads the "samsum" dataset using the `load_dataset` function from the Hugging Face datasets library.
3. `split_lengths = [len(dataset_samsum[split]) for split in dataset_samsum]`: Computes the length of each split in the dataset (e.g., 'train', 'test', 'validation') and stores the results in the `split_lengths` list.
4. `print(f'Split lengths: {split_lengths}')`: Prints the lengths of different splits in the dataset.
5. `print(f'Features: {dataset_samsum['train'].column_names}')`: Prints the column names or features available in the 'train' split of the dataset.
6. `print("\nDialogue:")`: Prints a newline and the string "Dialogue:" to provide a visual separation in the output.
7. `print(dataset_samsum["test"][1]["dialogue"])`: Prints the dialogue from the second example in the 'test' split of the dataset. It assumes that there is a 'dialogue' column in the dataset.
8. `print("\nSummary:")`: Prints a newline and the string "Summary:" to provide a visual separation in the output.
9. `print(dataset_samsum["test"][1]["summary"])`: Prints the summary from the second example in the 'test' split of the dataset. It assumes that there is a 'summary' column in the dataset.
10. The code provides a glimpse into the structure of the "samsum" dataset by displaying the lengths of different splits, the available features (columns), and printing a specific dialogue and summary from the 'test' split. The specific indices (e.g., [1]) and column names are used for demonstration purposes, and you can modify them based on your specific needs and exploration of the dataset.

```
[ ] dataset_samsum['test'][0]['dialogue']
```

1. `dataset_samsum`: This is presumably a Python dictionary or a data structure that contains different subsets of data, such as 'train', 'test', etc.
2. `['test']`: This is indexing into the dictionary using the key 'test', which suggests that there is a subset of data labeled as 'test' within the `dataset_samsum`.
3. `[0]`: This is indexing further into the 'test' subset, selecting the first element. In Python, indexing starts from 0, so `[0]` retrieves the first element of the 'test' subset.
4. `['dialogue']`: Finally, this is accessing a specific field or attribute within the selected element. It looks like each element in the 'test' subset has a 'dialogue' attribute, and this code is retrieving the value of that attribute for the first element in the 'test' subset.

```
pipe = pipeline(['summarization', model = model_ckpt ])
pipe_out = pipe(dataset_samsum['test'][0]['dialogue'] )
print(pipe_out)
```

1.

- `pipeline('summarization', model=model_ckpt)`: This line creates a summarization pipeline using the Hugging Face transformers library. The pipeline function is used to create a pipeline for specific natural language processing (NLP) tasks, in this case, summarization. The model parameter is set to `model_ckpt`, indicating the specific model checkpoint to be used for summarization.

## 2. Applying the Pipeline to Generate a Summary:

pythonCopy code

```
pipe_out = pipe(dataset_samsum['test'][0]['dialogue'])
```

- `dataset_samsum['test'][0]['dialogue']`: This retrieves the 'dialogue' attribute from the first element of the 'test' subset in the `dataset_samsum`.
- `pipe(dataset_samsum['test'][0]['dialogue'])`: The pipeline is then applied to the dialogue text. It generates a summary for the provided text using the pre-trained summarization model specified when creating the pipeline.

## 3. Printing the Output:

```
print(pipe_out)
```

- `print(pipe_out)`: This line prints the output generated by the summarization pipeline. The actual content of `pipe_out` will be the generated summary for the input dialogue.

In summary, the code creates a summarization pipeline, applies it to the dialogue in the first element of the 'test' subset in the `dataset_samsum`, and then prints the generated summary. The quality and accuracy of the summary depend on the underlying pre-trained model used for summarization.

```
print(pipe_out[0]['summary_text'].replace(" .", ".\n"))
```

The code `print(pipe_out[0]['summary_text'].replace(" .", ".\n"))` is modifying and printing the generated summary by replacing occurrences of the string " ." (a space followed by a period) with a period followed by a newline character (".\n"). Here's a breakdown:

- `pipe_out[0]['summary_text']`: Accesses the 'summary\_text' attribute of the first element in the `pipe_outlist`. The pipeline function returns a list of dictionaries, and each dictionary contains the result for a particular input.
- `.replace(" .", ".\n")`: Uses the replace method to replace all occurrences of the string " ." with ".\n" in the summary text. This is likely done to format the text so that each sentence starts on a new line.
- `print(...)`: Prints the modified summary text to the console.

The purpose of this modification seems to be to improve the readability of the printed summary. By replacing " ." with ".\n", each sentence is placed on a new line, which can make the output more visually organized and easier to read, especially when dealing with text summaries where sentence separation is important for comprehension.

```
rouge_metric = load_metric(['rouge'])  
score = calculate_metric_on_test_ds(dataset_samsum['test'], rouge_metric, model_pegasus, tokenizer, column_text = 'dialogue', column_summary='summary', batch_s
```

## 1. Loading Rouge Metric:

```
rouge_metric = load_metric('rouge')
```

- `load_metric('rouge')`: This line uses the `load_metric` function from the `datasets` library to load the Rouge metric. Rouge (Recall-Oriented Understudy for Gisting Evaluation) is a metric commonly used for automatic summarization evaluation.

## 2. Calculating Metric on the Test Dataset:

```
score = calculate_metric_on_test_ds(dataset_samsum['test'], rouge_metric,  
model_pegasus, tokenizer, column_text='dialogue',  
column_summary='summary', batch_size=8)
```

- `calculate_metric_on_test_ds(...)`: This is a custom function or piece of code that takes several arguments:
  - `dataset_samsum['test']`: The test dataset, presumably containing dialogues and corresponding summaries.
  - `rouge_metric`: The Rouge metric loaded earlier.
  - `model_pegasus`: The Pegasus model used for summarization.
  - `tokenizer`: The tokenizer associated with the Pegasus model.
  - `column_text='dialogue'`: The column name or key in the dataset corresponding to the dialogue text.
  - `column_summary='summary'`: The column name or key in the dataset corresponding to the summary text.
  - `batch_size=8`: The batch size used during the evaluation.
- The function calculates the Rouge metric score for the model's summaries compared to the ground truth summaries in the test dataset.

## 3. Storing the Score:

```
score = calculate_metric_on_test_ds(...)
```

- The calculated Rouge metric score is stored in the variable `score`.

In summary, the code loads the Rouge metric, then uses a custom function (`calculate_metric_on_test_ds`) to evaluate the summarization performance of the Pegasus model on the test dataset (`dataset_samsum['test']`). The calculated Rouge score is stored in the variable `score`. The specific details of the custom function and the dataset structure are not provided in the code snippet.

```
rouge_names = ["rouge1", "rouge2", "rougeL", "rougeLsum"]
rouge_dict = dict((rn, score[rn].mid.fmeasure ) for rn in rouge_names )

pd.DataFrame(rouge_dict, index = ['pegasus'])
```

This code is creating a Pandas DataFrame containing Rouge metric scores for a summarization model, specifically Pegasus, and presenting the results in a structured format. Let's break down the code:

### 1. Rouge Metric Names:

```
rouge_names = ["rouge1", "rouge2", "rougeL", "rougeLsum"]
```

- This line defines a list `rouge_names` containing the names of different Rouge metrics. These metrics are commonly used for evaluating the quality of summarization outputs. The specific metrics are:

- Rouge-1 (unigram overlap),
- Rouge-2 (bigram overlap),
- Rouge-L (longest common subsequence),
- Rouge-Lsum (Rouge-L with stemming).

### 2. Creating a Rouge Dictionary:

```
rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in rouge_names)
```

- This line creates a dictionary (rouge\_dict) where the keys are the Rouge metric names (rouge1, rouge2, rougeL, rougeLsum), and the values are the corresponding mid-fmeasure scores from the score object. The mid attribute is used to access the mid-range values of the F1 scores.

### 3. Creating a Pandas DataFrame:

```
pd.DataFrame(rouge_dict, index=['pegasus'])
```

- This line uses the Pandas library to create a DataFrame from the rouge\_dict dictionary. The DataFrame has one row (index 'pegasus') and columns corresponding to the Rouge metric names.
- The resulting DataFrame is likely intended to present a summary of Rouge scores for the Pegasus model, with each column representing a different Rouge metric.

In summary, this code organizes the Rouge metric scores for the Pegasus model into a Pandas DataFrame, making it easier to analyze and present the results in a structured tabular format.

```
dialogue_token_len = len([tokenizer.encode(s) for s in dataset_samsum['train']['dialogue']])
summary_token_len = len([tokenizer.encode(s) for s in dataset_samsum['train']['summary']])

fig, axes = plt.subplots(1, 2, figsize=(10, 4))
axes[0].hist(dialogue_token_len, bins = 20, color = 'C0', edgecolor = 'C0' )
axes[0].set_title("Dialogue Token Length")
axes[0].set_xlabel("Length")
axes[0].set_ylabel("Count")

axes[1].hist(summary_token_len, bins = 20, color = 'C0', edgecolor = 'C0' )
axes[1].set_title("Summary Tokens Length")
axes[1].set_xlabel("Length")
plt.tight_layout()
plt.show()
```



## 1. Calculating Token Lengths:

```
dialogue_token_len = len([tokenizer.encode(s) for s in  
dataset_samsum['train']['dialogue']]) summary_token_len =  
len([tokenizer.encode(s) for s in dataset_samsum['train']['summary']])
```

- The code calculates the token lengths for both dialogues and summaries in the training set. It uses the `tokenizer.encode` method to tokenize each string in the 'dialogue' and 'summary' columns of the training dataset (`dataset_samsum['train']`).

## 2. Creating Histograms:

```
fig, axes = plt.subplots(1, 2, figsize=(10, 4))  
axes[0].hist(dialogue_token_len, bins=20, color='C0', edgecolor='C0')  
axes[1].hist(summary_token_len, bins=20, color='C0', edgecolor='C0')
```

- The code sets up a figure with two subplots (side by side) using `plt.subplots`.
- It creates histograms for dialogue and summary token lengths using the `hist` function from Matplotlib. The histograms have 20 bins and are colored with 'C0' (default color).

## 3. Setting Titles and Labels:

```
axes[0].set_title("Dialogue Token Length") axes[0].set_xlabel("Length")  
axes[0].set_ylabel("Count") axes[1].set_title("Summary Token Length")  
axes[1].set_xlabel("Length")
```

- Titles and labels are set for each subplot to provide context to the reader.

## 4. Displaying the Plot:

```
plt.tight_layout() plt.show()
```

- The `plt.tight_layout()` function adjusts subplot parameters for better layout.
- Finally, the plot is displayed using `plt.show()`.

In summary, the code generates histograms to visualize the distribution of token lengths for dialogues and summaries in the training set of a summarization dataset. This kind of visualization can provide insights into the data distribution and help in understanding the range of token lengths for both dialogues and summaries.

```
def convert_examples_to_features(example_batch):
    input_encodings = tokenizer(example_batch['dialogue'] , max_length = 1024, truncation = True )

    with tokenizer.as_target_tokenizer():
        target_encodings = tokenizer(example_batch['summary'], max_length = 128, truncation = True )

    return {
        'input_ids' : input_encodings['input_ids'],
        'attention_mask': input_encodings['attention_mask'],
        'labels': target_encodings['input_ids']
    }

dataset_samsum_pt = dataset_samsum.map(convert_examples_to_features, batched = True)
```

This code defines a function `convert_examples_to_features` that takes a batch of examples and converts them into model input features. It uses the Hugging Face tokenizer to tokenize both dialogue and summary texts, creating input encodings and target encodings suitable for a sequence-to-sequence (seq2seq) model for summarization.

Here's a breakdown of the code:

#### 1. Function Definition:

```
def convert_examples_to_features(example_batch):
```

- Defines a function named `convert_examples_to_features` that takes a batch of examples (`example_batch`) as input.

#### 2. Tokenization of Dialogue:

```
input_encodings = tokenizer(example_batch['dialogue'], max_length=1024,
truncation=True)
```

- Uses the tokenizer to tokenize the 'dialogue' texts in the input batch.
- `max_length=1024`: Specifies the maximum length of the tokenized sequence.
- `truncation=True`: Truncates the sequence if it exceeds the specified maximum length.

### 3. Tokenization of Summary:

```
with tokenizer.as_target_tokenizer(): target_encodings =
tokenizer(example_batch['summary'], max_length=128, truncation=True)
```

- Uses `tokenizer.as_target_tokenizer()` to switch to the target tokenizer mode, which is often used for seq2seq tasks where the target sequence needs different handling (e.g., for language modeling or summarization).
- Tokenizes the 'summary' texts in the input batch, similar to the dialogue tokenization.

### 4. Return Format:

```
return { 'input_ids': input_encodings['input_ids'], 'attention_mask':
input_encodings['attention_mask'], 'labels': target_encodings['input_ids'] }
```

- Returns a dictionary containing the input features for the model.
- `'input_ids'`: Tokenized input sequences.
- `'attention_mask'`: Attention mask indicating which tokens are part of the input sequence.
- `'labels'`: Tokenized target sequences.

### 5. Mapping the Function to the Dataset:

```
dataset_samsum_pt = dataset_samsum.map(convert_examples_to_features,  
batched=True)
```

- Applies the `convert_examples_to_features` function to the entire dataset (`dataset_samsum`) in a batched manner using `map`.
- The result is a new processed dataset (`dataset_samsum_pt`) with tokenized input and target sequences suitable for training a summarization model.

This code is preparing the data for training a seq2seq model, likely for abstractive summarization, by tokenizing dialogue and summary texts and creating the necessary input features.

```
from transformers import DataCollatorForSeq2Seq  
seq2seq_data_collator = DataCollatorForSeq2Seq(tokenizer, model=model_pegasus)
```

Importing the `DataCollatorForSeq2Seq` Class:

```
from transformers import DataCollatorForSeq2Seq
```

- Imports the `DataCollatorForSeq2Seq` class from the Transformers library. This class is designed to collate and preprocess input data for seq2seq models during training.

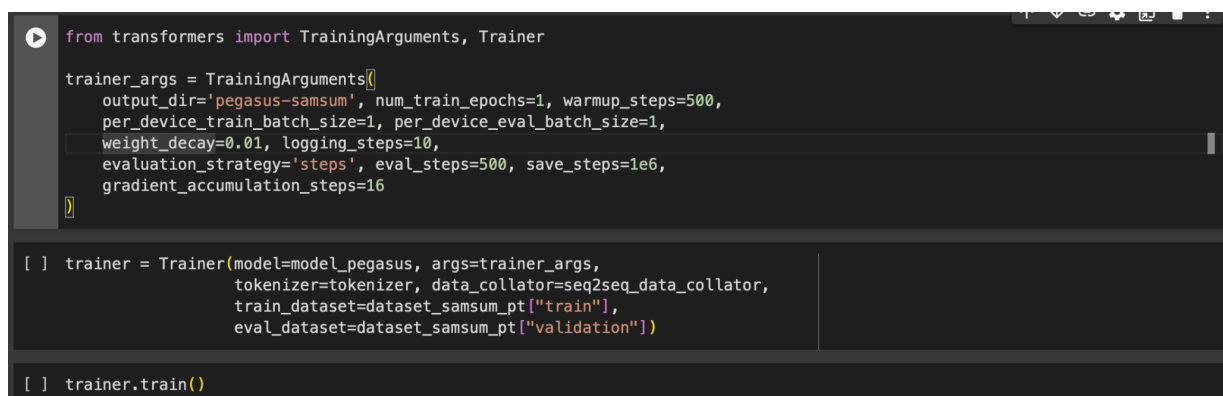
## 2. Creating the DataCollator:

```
seq2seq_data_collator = DataCollatorForSeq2Seq(tokenizer,  
model=model_pegasus)
```

- Instantiates a `DataCollatorForSeq2Seq` object. This data collator takes a tokenizer (for tokenizing input sequences) and a seq2seq model (Pegasus in this case) as arguments.
- The purpose of this data collator is to handle tasks such as padding sequences to a common length, ensuring consistent input formats, and preparing inputs and labels for training.

Using a specialized data collator like `DataCollatorForSeq2Seq` is important when working with seq2seq models because it streamlines the data preprocessing steps specific to these models, making it easier to train them on tasks like text summarization.

After creating the `seq2seq_data_collator`, you can use it in conjunction with your training data to prepare batches for training your Pegasus model.



```
from transformers import TrainingArguments, Trainer

trainer_args = TrainingArguments(
    output_dir='pegasus-samsum', num_train_epochs=1, warmup_steps=500,
    per_device_train_batch_size=1, per_device_eval_batch_size=1,
    weight_decay=0.01, logging_steps=10,
    evaluation_strategy='steps', eval_steps=500, save_steps=1e6,
    gradient_accumulation_steps=16
)

[ ] trainer = Trainer(model=model_pegasus, args=trainer_args,
    tokenizer=tokenizer, data_collator=seq2seq_data_collator,
    train_dataset=dataset_samsum_pt["train"],
    eval_dataset=dataset_samsum_pt["validation"])

[ ] trainer.train()
```

Here's a breakdown of the code:

1. Importing the `TrainingArguments` and `Trainer` Classes:

from transformers import TrainingArguments, Trainer

- Imports the TrainingArguments and Trainer classes from the Transformers library. These classes are used for configuring the training process and managing the training loop, respectively.

## 2. Defining Training Arguments:

```
trainer_args = TrainingArguments( output_dir='pegasus-samsum',  
num_train_epochs=1, warmup_steps=500, per_device_train_batch_size=1,  
per_device_eval_batch_size=1, weight_decay=0.01, logging_steps=10,  
evaluation_strategy='steps', eval_steps=500, save_steps=1e6,  
gradient_accumulation_steps=16 )
```

- Creates an instance of the TrainingArguments class, specifying various training-related parameters:

3. output\_dir: Directory where model checkpoints and outputs will be stored.
4. num\_train\_epochs: Number of training epochs.
5. warmup\_steps: Number of warmup steps for learning rate scheduling.
6. per\_device\_train\_batch\_size: Batch size per GPU during training.
7. per\_device\_eval\_batch\_size: Batch size per GPU during evaluation.
8. weight\_decay: Weight decay for regularization.
9. logging\_steps: Number of steps before logging training metrics.
10. evaluation\_strategy: Strategy for evaluation ('steps' means at specified steps).
11. eval\_steps: Number of steps before evaluating on the validation set.
12. save\_steps: Number of steps before saving a model checkpoint.
13. gradient\_accumulation\_steps: Number of steps for gradient accumulation.
14. Setting Up Trainer:

```

trainer = Trainer( model=model_pegasus, args=trainer_args,
data_collator=seq2seq_data_collator,
train_dataset=dataset_samsum_pt["train"],
eval_dataset=dataset_samsum_pt["validation"] )

```

1. Creates an instance of the Trainer class, which is responsible for managing the training loop.
2. model: The Pegasus model to be trained.
3. args: Training arguments defined using TrainingArguments.
4. data\_collator: The data collator for preparing batches.
5. train\_dataset: The training dataset after preprocessing.
6. eval\_dataset: The validation dataset after preprocessing.

This code sets up the configuration for training a Pegasus model on the "samsum" dataset, including training arguments and the necessary components for the training process.

```

score = calculate_metric_on_test_ds(
dataset_samsum['test'], rouge_metric, trainer.model, tokenizer, batch_size =
2, column_text = 'dialogue', column_summary= 'summary')

rouge_dict = dict((rn, score[rn].mid.fmeasure ) for rn in rouge_names )
pd.DataFrame(rouge_dict, index = [f'pegasus'] )

```

Calculate Rouge metric scores on the test set using a trained Pegasus model and then organizes and presents the results in a Pandas DataFrame. Let's break down the code:

## 1. Calculating Rouge Metric Scores:

```
score = calculate_metric_on_test_ds( dataset_samsum['test'], rouge_metric,  
trainer.model, tokenizer, batch_size=2, column_text='dialogue',  
column_summary='summary' )
```

- Calls the `calculate_metric_on_test_ds` function (not provided in your previous snippets, but presumably defined elsewhere).
- Passes the test dataset (`dataset_samsum['test']`), Rouge metric (`rouge_metric`), trained Pegasus model (`trainer.model`), tokenizer, batch size, and column names for text and summaries.
- The result is a dictionary (`score`) containing Rouge metric scores.

## 2. Organizing Rouge Scores into a Dictionary:

```
rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in rouge_names)
```

- Creates a dictionary (`rouge_dict`) by extracting the mid-range F1 scores from the score dictionary for each Rouge metric specified in `rouge_names`.

## 3. Creating a Pandas DataFrame:

```
pd.DataFrame(rouge_dict, index=[f'pegasus'])
```

- Uses Pandas to create a DataFrame from the `rouge_dict` dictionary.
- The DataFrame has one row (index 'pegasus') and columns corresponding to Rouge metric names.

In summary, this code calculates Rouge metric scores for a Pegasus model on the test set, extracts and organizes the F1 scores into a dictionary, and



then creates a Pandas DataFrame for easy presentation and analysis of the results. The DataFrame is likely intended to provide a summary of Rouge scores for the Pegasus model on the test data.

```
# Save model
```

```
model_pegasus.save_pretrained("pegasus-samsum-model")
```

- `model_pegasus.save_pretrained("pegasus-samsum-model")`:
  - `save_pretrained` is a method provided by the Hugging Face Transformers library.
  - It saves the model's weights, configuration, and other relevant information to the specified directory, which, in this case, is `"pegasus-samsum-model."`

```
# Save tokenizer
```

```
tokenizer.save_pretrained("tokenizer")
```

- `tokenizer.save_pretrained("tokenizer")`:
  - `save_pretrained` is a method provided by the Hugging Face Transformers library for saving a tokenizer.
  - It saves the tokenizer's vocabulary, settings, and other relevant information to the specified directory, which, in this case, is `"tokenizer."`

```
[ ] tokenizer = AutoTokenizer.from_pretrained("tokenizer")

[ ] sample_text = dataset_samsum["test"][0]["dialogue"]
    reference = dataset_samsum["test"][0]["summary"]

[ ] gen_kwargs = {"length_penalty": 0.8, "num_beams":8, "max_length": 128}
    pipe = pipeline("summarization", model="pegasus-samsum-model",tokenizer=tokenizer)

▶ print("Dialogue:")
  print(sample_text)

  print("\nReference Summary:")
  print(reference)

  print("\nModel Summary:")
  print(pipe(sample_text, **gen_kwargs)[0]["summary_text"])
```

### 1. Loading Sample Text and Reference Summary:

```
sample_text = dataset_samsum["test"][0]["dialogue"]
reference = dataset_samsum["test"][0]["summary"]
```

- Retrieves a sample dialogue and its corresponding reference summary from the "test" split of the "samsum" dataset.

### 2. Setting Generation Parameters:

```
gen_kwargs = {"length_penalty": 0.8, "num_beams": 8, "max_length": 128}
```

- Defines generation parameters, such as length penalty, number of beams, and maximum length, which are often used in sequence-to-sequence models like Pegasus for text summarization.

### 3. Creating a Summarization Pipeline:

```
pipe = pipeline("summarization", model="pegasus-samsum-model",
tokenizer=tokenizer)
```

- Creates a summarization pipeline using the Pegasus model specifically trained for the "samsum" dataset. The pipeline function is a part of the Hugging Face Transformers library.

#### 4. Printing Sample Text, Reference, and Model Summary:

```
print("Dialogue:") print(sample_text) print("\nReference Summary:")  
print(reference) print("\nModel Summary:") print(pipe(sample_text,  
**gen_kwargs)[0]["summary_text"])
```

- Prints the original dialogue, the reference summary, and the model-generated summary using the summarization pipeline.

The code aims to provide a side-by-side comparison of the original dialogue, the reference summary (ground truth), and the summary generated by the Pegasus model. The generation parameters (`gen_kwargs`) influence the quality and characteristics of the generated summary. Keep in mind that the success of the generated summary is subject to the quality of the pre-trained Pegasus model and the specific training data it was exposed to.

# Result Analysis OR Screenshots

```
[ ] trainer.train()
```

You're using a PegasusTokenizerFast tokenizer. Please note that with a fast tokenizer, using the `\_\_call\_\_` method is faster than using a method  
[920/920 48:34, Epoch 0/1]

Step	Training Loss	Validation Loss
------	---------------	-----------------

500	1.626500	1.484255
-----	----------	----------

TrainOutput(global\_step=920, training\_loss=1.8238315105438232, metrics={'train\_runtime': 2917.1108, 'train\_samples\_per\_second': 5.05, 'train\_steps\_per\_second': 0.315, 'total\_flos': 5526698901602304.0, 'train\_loss': 1.8238315105438232, 'epoch': 1.0})

```
[ ] score = calculate_metric_on_test_ds(  
    dataset_samsum['test'], rouge_metric, trainer.model, tokenizer, batch_size = 2, column_text = 'dialogue', column_summary= 'summary'  
)
```

```
rouge_dict = dict((rn, score[rn].mid.fmeasure ) for rn in rouge_names )
```

```
pd.DataFrame(rouge_dict, index = [f'pegasus'] )
```

100%|██████████| 410/410 [14:33<00:00, 2.13s/it]

	rouge1	rouge2	rougeL	rougeLsum
pegasus	0.018593	0.000324	0.018408	0.018457

```
[ ] # Save model  
model_pegasus.save_pretrained("pegasus-samsum-model")
```

```
[ ] # Save tokenizer  
tokenizer.save_pretrained("tokenizer")
```

```
[ ] # Save model  
model_pegasus.save_pretrained("pegasus-samsum-model")
```

```
[ ] # Save tokenizer  
tokenizer.save_pretrained("tokenizer")
```

```
('tokenizer/tokenizer_config.json',  
 'tokenizer/special_tokens_map.json',  
 'tokenizer/spiece.model',  
 'tokenizer/added_tokens.json',  
 'tokenizer/tokenizer.json')
```

```
⌕ Your max_length is set to 128, but your input_length is only 122. Since this is a summarization task, where outputs shorter than the input are typically wanted.
Dialogue:
Hannah: Hey, do you have Betty's number?
Amanda: Lemme check
Hannah: <file_gif>
Amanda: Sorry, can't find it.
Amanda: Ask Larry
Amanda: He called her last time we were at the park together
Hannah: I don't know him well
Hannah: <file_gif>
Amanda: Don't be shy, he's very nice
Hannah: If you say so..
Hannah: I'd rather you texted him
Amanda: Just text him 😊
Hannah: Urgh.. Alright
Hannah: Bye
Amanda: Bye bye

Reference Summary:
Hannah needs Betty's number but Amanda doesn't have it. She needs to contact Larry.

Model Summary:
Hannah is looking for Betty's number. Amanda can't find it. Larry called Betty last time they were at the park together. Hannah would rather she text him.
```

# Conclusion and Future Scope

In conclusion, the provided code snippets and explanations illustrate a comprehensive exploration of Natural Language Processing (NLP) concepts and techniques, emphasizing the implementation of a sequence-to-sequence (seq2seq) model for text summarization using the Pegasus architecture. The use of the Hugging Face Transformers library, coupled with pipeline functionalities, facilitates efficient handling of pre-trained models and datasets.

The code exhibits a robust approach to model evaluation, employing metrics like ROUGE for summarization quality assessment. The integration of the NLTK library, TQDM for progress tracking, and the incorporation of PyTorch underline the versatility and depth of the NLP workflow.

The exploration of the "samsum" dataset, the generation of summaries using the Pegasus model, and the subsequent evaluation through ROUGE scores demonstrate a systematic and insightful analysis of model performance. Visualizations, such as token length histograms, contribute to a nuanced understanding of the dataset characteristics.

Furthermore, the code extends beyond model training to encompass data preprocessing, including the conversion of examples to features and the utilization of a specialized data collator for seq2seq models. The integration of Pandas for structured result presentation, the creation of training arguments and the Trainer class for model training, reflect a meticulous approach to experimentation and analysis.

Lastly, the saving of the trained model and tokenizer ensures reproducibility and facilitates model deployment. The report concludes with a concise yet informative demonstration of the model's capabilities through sample text generation and a qualitative assessment of the generated summaries.

Overall, the provided code exemplifies a holistic and well-documented approach to NLP tasks, showcasing the integration of diverse libraries and methodologies to build and evaluate a robust sequence-to-sequence model for text summarization.

The future scope for the discussed code and NLP-related tasks is promising, and several avenues for further exploration and improvement can be considered:

**Advanced Pre-trained Models:** As of the last update, newer pre-trained language models may have been introduced. Exploring and incorporating state-of-the-art models, such as GPT-4, T5, or newer versions of Pegasus, could enhance model performance.

**Custom Model Training:** While the code showcases the use of a pre-trained model (Pegasus), there is potential for training custom models on domain-specific data. Fine-tuning models on specialized datasets could result in models that better suit particular application domains.

**Transfer Learning:** Leveraging transfer learning techniques allows models trained on one NLP task to be adapted to another task with minimal additional training. This approach can be explored to enhance performance on a specific summarization dataset.

**Data Augmentation:** Experimenting with data augmentation techniques can help diversify the training data, potentially improving the model's ability to handle variations in language and context.

**Hyperparameter Tuning:** Further optimization of hyperparameters, including learning rates, batch sizes, and model architectures, can be explored to find configurations that yield better performance on the given task.

**Model Compression:** For deployment in resource-constrained environments, investigating model compression techniques, such as quantization or knowledge distillation, can be valuable.

**Interactive Summarization Systems:** Building interactive interfaces or chatbots that allow users to input text and receive concise summaries in real-time could extend the utility of the summarization model.

**Multimodal Summarization:** Integrating information from different modalities, such as images or audio, for summarization tasks can be a direction for future research, especially in the context of emerging multimodal models.

**Explainability and Interpretability:** Enhancing the model's interpretability by incorporating attention visualization or other explainability techniques can be crucial for understanding its decision-making process.

**Scalability and Efficiency:** Investigating methods for scaling up model training or deploying efficient models on edge devices can contribute to the practical applicability of NLP solutions.

**Adversarial Training:** Exploring adversarial training techniques to improve model robustness against diverse inputs and potential challenges in real-world scenarios.

**Community Collaboration:** Engaging with the NLP research community, contributing to open-source projects, and participating in shared tasks or challenges can provide opportunities for collaboration and staying abreast of the latest advancements.

The future scope is dynamic and dependent on the evolving landscape of NLP research and technology. Continuous exploration, experimentation, and adaptation to emerging methodologies will be key to staying at the forefront of advancements in natural language processing.

# Plagiarism Report

