

▼ Text Summarization Project

Introduction

In the era of information overload, the ability to distill essential insights from large volumes of text is more crucial than ever. This is where the field of text summarization comes into play. Text summarization is a branch of Natural Language Processing (NLP) that involves condensing a larger body of text into a short, coherent summary while preserving its key informational elements and overall meaning.

The primary goal of this project is to build a text summarization model that can take a lengthy piece of text and generate a concise summary, much like a human would. This has wide-ranging applications in numerous fields such as journalism, where it can be used to generate news digests, in academia for literature reviews, or in business for summarizing reports and meetings.

One popular example of a text summarization tool is Quillbot, which uses advanced NLP techniques to paraphrase and summarize text. Similarly, our project aims to create a model that can understand the context and extract the salient points from a piece of text, thereby saving the reader's time and effort.

For instance, given a long article about climate change, our model should be able to generate a summary like: "The article discusses the increasing threat of climate change, highlighting the rise in global temperatures and sea levels. It emphasizes the need for urgent action, suggesting renewable energy and sustainable practices as potential solutions."

In the following sections, we will walk through the steps of building such a model, from data collection and preprocessing to model training and evaluation. We will also discuss the challenges faced during the project and how we overcame them. Let's dive in!

▼ Objective

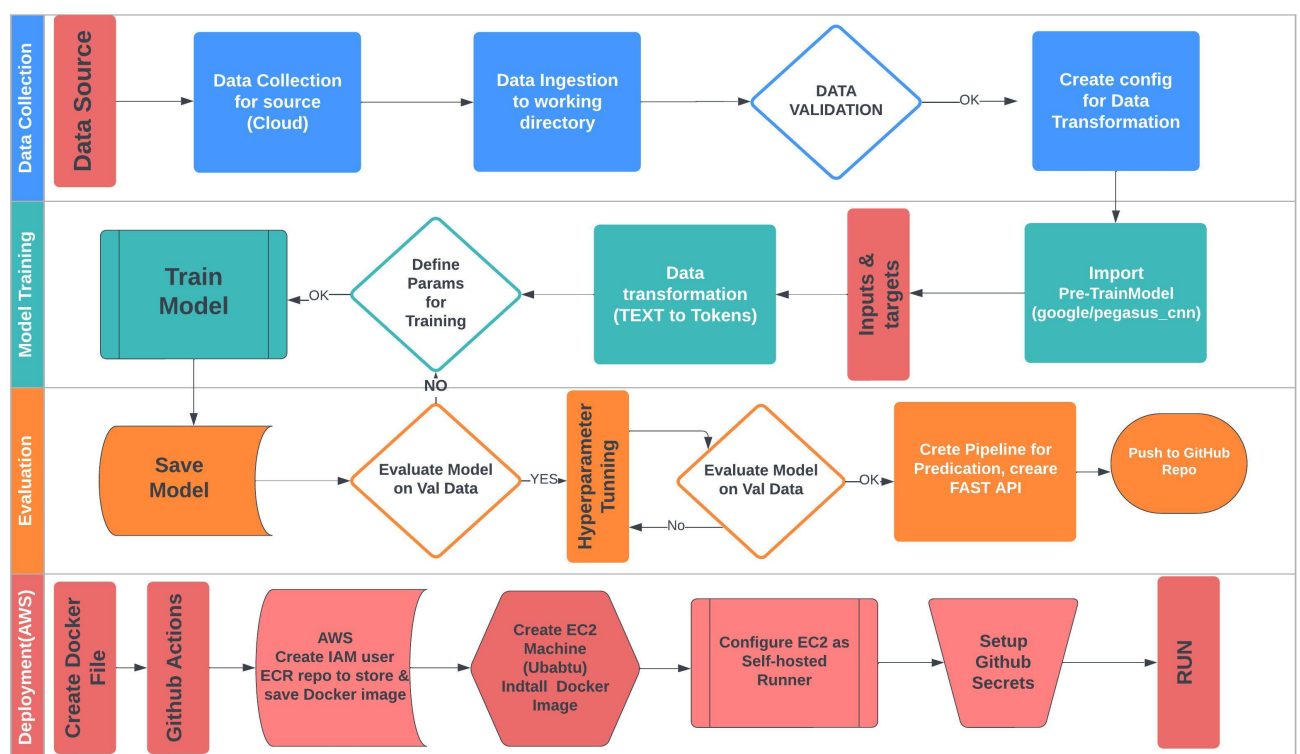
The primary objective of this project is to design and implement an effective text summarization model that can condense lengthy textual information into a concise and coherent summary. The aim is to retain the key points and overall context of the original text in the summary, thereby providing a quick and accurate understanding of the content without the need to read through the entire text.

In the context of this project, I am particularly interested in exploring the application of advanced Natural Language Processing (NLP) techniques and machine learning algorithms for this task. I aim to build a model that can handle a variety of text types, ranging from news articles and research papers to blog posts and book chapters.

Another key objective is to ensure that the generated summaries are not only short and informative but also grammatically correct and readable. The model should be able to generate summaries that are smooth, coherent, and can stand on their own as a comprehensive reflection of the original text.

Furthermore, I aim to evaluate the performance of my model rigorously and objectively, using established evaluation metrics in the field of NLP. This will allow me to understand the strengths and weaknesses of my model and guide future improvements.

Ultimately, the goal is to contribute to the ongoing efforts in the field of NLP to make information more accessible and manageable in the face of growing data. By creating an effective text summarization model, I hope to provide a tool that can save time and effort for anyone who needs to understand large volumes of text quickly and efficiently.



▼ Data Collection

For this project, I used the [SAMSum Corpus](#), a dataset that contains approximately 16,000 messenger-like conversations along with their summaries. These conversations were created and written down by linguists fluent in English, reflecting a variety of topics, styles, and registers. The dataset was prepared by Samsung R&D Institute Poland and is distributed for research purposes.

Data Preparation

The SAMSum Corpus is already structured and annotated, which simplifies the data preparation process. However, it's still necessary to preprocess the data for the specific requirements of the

model. This includes tokenizing the text and converting it into a format that can be fed into the model.

Model Selection

For this project, I chose to use the pre-trained PEGASUS model provided by Google. PEGASUS (Pre-training with Extracted Gap-sentences for Abstractive SUMmarization Sequence-to-sequence) is a state-of-the-art model for abstractive text summarization. The specific version of the model I used is '[google/pegasus-cnn_dailymail](#)', which has been trained on both the C4 and HugeNews datasets.

Model Training

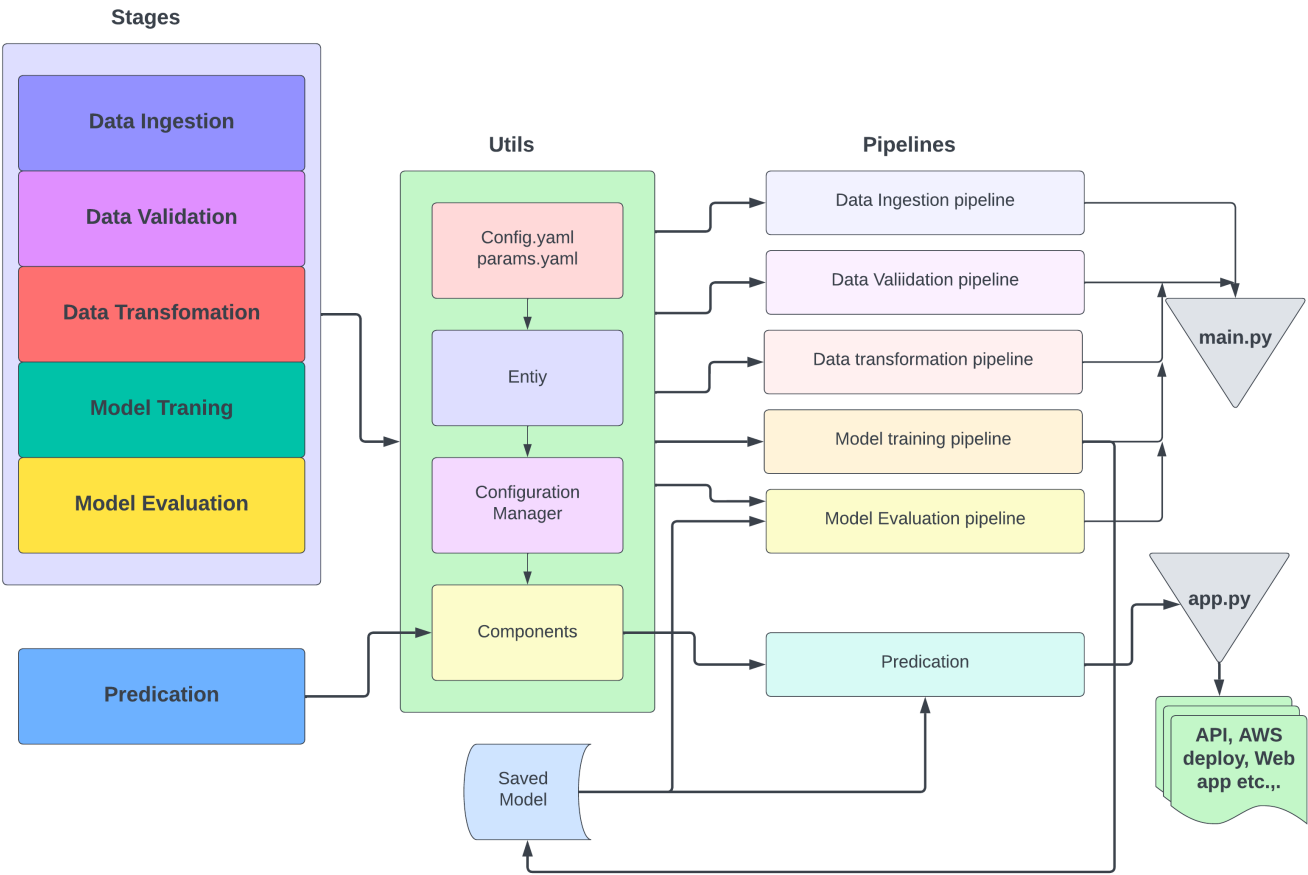
The pre-trained PEGASUS model was then fine-tuned on the SAMSum Corpus. Fine-tuning is a process where the pre-trained model is further trained on a specific task (in this case, text summarization), allowing it to adapt to the specific characteristics of the task.

Model Evaluation

After the model was fine-tuned, it was evaluated on a separate test set from the SAMSum Corpus. The performance of the model was measured using established evaluation metrics for text summarization, such as ROUGE and BLEU scores.

Model Deployment

Once the model was trained and evaluated, it was deployed using AWS and GitHub Actions. AWS provides a robust platform for hosting machine learning models, while GitHub Actions allows for automated deployment processes. This ensures that any updates to the model or the code are automatically reflected in the deployed application.



```
1 !nvidia-smi
```

```
Fri Jul 28 19:51:16 2023
```

NVIDIA-SMI 525.105.17 Driver Version: 525.105.17 CUDA Version: 12.0									
GPU Name Persistence-M				Bus-Id	Disp.A	Volatile Uncorr. ECC			
Fan	Temp	Perf	Pwr:Usage/Cap		Memory-Usage	GPU-Util	Compute	M.	
MIG M.									
0	Tesla T4	Off	00000000:00:04.0	Off		0			
N/A	71C	P8	11W / 70W		0MiB / 15360MiB	0%	Default		N/A

Processes:							
GPU	GI	CI	PID	Type	Process name	GPU Memory	
ID	ID					Usage	
No running processes found							

```
1 !pip install transformers[sentencepiece] datasets sacrebleu rouge_score py7zr -q
2
```

```
1
2 !pip install --upgrade accelerate -q
```

```
3 !pip uninstall -y transformers accelerate -q
4 !pip install transformers accelerate -q
```

▼ Dependencies and Setup

Before we start with the actual model training and evaluation, we need to set up our environment and download the necessary dependencies. Here are the libraries and modules we will be using:

- **Transformers:** This library, developed by Hugging Face, provides thousands of pre-trained models to perform tasks on texts such as classification, information extraction, summarization, etc. We will be using it to access the pre-trained PEGASUS model.
- **Datasets:** Also developed by Hugging Face, this library provides a simple API to download and preprocess datasets. We will be using it to download the SAMSum Corpus.
- **Matplotlib:** This is a comprehensive library for creating static, animated, and interactive visualizations in Python. We will be using it to plot the training progress.
- **Pandas:** This library provides high-performance, easy-to-use data structures and data analysis tools for Python. We will be using it to manipulate our data.
- **NLTK:** The Natural Language Toolkit (NLTK) is a platform used for building Python programs to work with human language data. We will be using it for tokenization.
- **TQDM:** This library provides a fast, extensible progress bar for Python and CLI. We will be using it to visualize the progress of our loops.
- **Torch:** PyTorch is an open-source machine learning library based on the Torch library. We will be using it as our main library to train our model.
- **Warnings:** This is a standard Python module for warning control. We will be using it to ignore any warning messages to keep our output clean.

```
1 from transformers import pipeline, set_seed
2 from datasets import load_dataset, load_from_disk
3 import matplotlib.pyplot as plt
4 from datasets import load_dataset
5 import pandas as pd
6 from datasets import load_dataset, load_metric
7
8 from transformers import AutoModelForSeq2SeqLM, AutoTokenizer
9
10 import nltk
11 from nltk.tokenize import sent_tokenize
12
13 from tqdm import tqdm
14 import torch
15
16 nltk.download("punkt")
17
18
```

```
19 import warnings
20
21 warnings.filterwarnings("ignore")
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
```

```
[nltk_data]   Package punkt is already up-to-date!
```

▼ Setting up the Device

Before we start with the model training, we need to set up our device configuration. In PyTorch, we need to set up our device as either CPU or CUDA (which stands for Compute Unified Device Architecture, a parallel computing platform and application programming interface model created by NVIDIA). If a GPU is available, PyTorch will use it by default, otherwise, it will use the CPU.

```
1
2 from transformers import AutoModelForSeq2SeqLM, AutoTokenizer
3
4 device = "cuda" if torch.cuda.is_available() else "cpu"
5 device
6
```

```
'cuda'
```

▼ Loading the Pre-trained Model and Tokenizer

We will be using the PEGASUS model pre-trained on the CNN/DailyMail dataset. The model checkpoint is available on the Hugging Face model hub under the name 'google/pegasus-cnn_dailymail'.

We first load the tokenizer associated with the model. The tokenizer is responsible for preprocessing the text for the model. This includes steps like tokenization, which is the process of converting the text into tokens (smaller parts like words or subwords), and encoding, which is the process of converting these tokens into numbers that the model can understand.

Next, we load the pre-trained PEGASUS model. We specify that the model should be loaded onto our device (either the CPU or GPU, depending on availability).

```
1
2 model_ckpt = "google/pegasus-cnn_dailymail"
3
4 tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
5
6 model_pegasus = AutoModelForSeq2SeqLM.from_pretrained(model_ckpt).to(device)
```

```
Some weights of PegasusForConditionalGeneration were not initialized from the model checkpoint
You should probably TRAIN this model on a down-stream task to be able to use it for inference.
```

▼ Downloading and Unzipping the Dataset

The SAMSum dataset is used for this project. It is available on GitHub and can be downloaded directly. After downloading, the dataset is unzipped to access the data files.

```

1
2 #download & unzip data
3
4 !wget https://github.com/nani2357/My_dataset_repo/raw/main/samsumdata.zip
5 !unzip summarizer-data.zip

--2023-07-28 19:53:08-- https://github.com/entbappy/Branching-tutorial/raw/master/s
Resolving github.com (github.com)... 20.205.243.166
Connecting to github.com (github.com)|20.205.243.166|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/entbappy/Branching-tutorial/master/summar
--2023-07-28 19:53:08-- https://raw.githubusercontent.com/entbappy/Branching-tutoria
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 1
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|
HTTP request sent, awaiting response... 200 OK
Length: 7903594 (7.5M) [application/zip]
Saving to: 'summarizer-data.zip.1'

summarizer-data.zip 100%[=====>] 7.54M --.-KB/s in 0.08s

2023-07-28 19:53:09 (92.0 MB/s) - 'summarizer-data.zip.1' saved [7903594/7903594]

Archive: summarizer-data.zip
replace samsum-test.csv? [y]es, [n]o, [A]ll, [N]one, [r]ename:

```

The dataset is loaded from the disk using the `load_from_disk` function from the `datasets` library. This function reads a dataset that was previously saved using the `save_to_disk` function.

```

1
2 dataset_samsum = load_from_disk('samsum_dataset')
3 dataset_samsum
4

DatasetDict({
  train: Dataset({
    features: ['id', 'dialogue', 'summary'],
    num_rows: 14732
  })
  test: Dataset({
    features: ['id', 'dialogue', 'summary'],
    num_rows: 819
  })
  validation: Dataset({
    features: ['id', 'dialogue', 'summary'],
    num_rows: 818
  })
})

```

```
    })  
  })
```

▼ Exploring the Dataset

To understand the dataset better, the length of each split (train, test, validation) is calculated and the features of the dataset are printed. The dataset consists of dialogues and their corresponding summaries.

The dialogue and summary of a sample from the test set are also printed to get a sense of what the data looks like.

```
1 split_lengths = [len(dataset_samsum[split])for split in dataset_samsum]
2
3 print(f"Split lengths: {split_lengths}")
4 print(f"Features: {dataset_samsum['train'].column_names}")
5 print("\nDialogue:")
6
7 print(dataset_samsum["test"][1]["dialogue"])
8
9 print("\nSummary:")
10
11 print(dataset_samsum["test"][1]["summary"])
12
```

```
Split lengths: [14732, 819, 818]
Features: ['id', 'dialogue', 'summary']
```

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.

▼ Preprocessing the Data

The data needs to be preprocessed before it can be fed into the model. This involves converting the dialogues and summaries into a format that the model can understand.

A function `convert_examples_to_features` is defined for this purpose. This function takes a batch of examples and performs the following steps:

1. It tokenizes the dialogues using the tokenizer's `__call__` method. The dialogues are truncated to a maximum length of 1024 tokens.
2. It tokenizes the summaries in a similar way, but with a maximum length of 128 tokens. The tokenizer is switched to target mode using the `as_target_tokenizer` context manager. This is because the summaries are the targets that the model will be trained to predict.
3. It returns a dictionary containing the input IDs, attention masks, and labels. The input IDs and attention masks are derived from the tokenized dialogues, and the labels are derived from the tokenized summaries.

```

1 def convert_examples_to_features(example_batch):
2     input_encoding = tokenizer(example_batch['dialogue'],
3                               max_length = 1024,
4                               truncation=True
5                               )
6
7     with tokenizer.as_target_tokenizer():
8         target_encoding = tokenizer(example_batch['summary'],
9                                    max_length=128,
10                                   truncation=True
11                                   )
12
13     return {
14         'input_ids':input_encoding['input_ids'],
15         'attention_mask':input_encoding['attention_mask'],
16         'labels' : target_encoding['input_ids']
17     }

```

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

```
Map: 819/819 [00:00<00:00, 1820.76
100% examples/s]
```

```
1 dataset_samsum_pt["train"]
```

```

Dataset({
  features: ['id', 'dialogue', 'summary', 'input_ids', 'attention_mask',
'labels'],
  num_rows: 14732
})

```

▼ Training the Model

The model is trained using the Hugging Face's `Trainer` class. The `Trainer` requires a number of arguments:

1. `model`: The model to be trained, which in this case is the Pegasus model.
2. `args`: Training arguments that specify the training parameters. The `TrainingArguments` class is used to create the training arguments.
3. `data_collator`: The data collator is responsible for batching the data. The `DataCollatorForSeq2Seq` class is used to create the data collator.
4. `tokenizer`: The tokenizer used for preprocessing the data.
5. `train_dataset` and `eval_dataset`: The training and validation datasets.

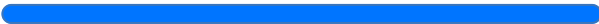
The `Trainer` is then used to train the model with the `train` method.

```
1 #training
2
3 from transformers import DataCollatorForSeq2Seq
4
5 seq2seq_data_collator = DataCollatorForSeq2Seq(tokenizer, model=model_pegasus)
```

```
1 from transformers import TrainingArguments, Trainer
2
3 trainer_args = TrainingArguments(
4     output_dir='pegasus-samsum', num_train_epochs=1, warmup_steps=500,
5     per_device_train_batch_size=1, per_device_eval_batch_size=1,
6     weight_decay=0.01, logging_steps=10,
7     evaluation_strategy='steps', eval_steps=500, save_steps=1e6,
8     gradient_accumulation_steps=16
9 )
10
```

```
1 trainer = Trainer(model=model_pegasus, args=trainer_args,
2                   tokenizer=tokenizer, data_collator=seq2seq_data_collator,
3                   train_dataset=dataset_samsum_pt["test"],
4                   eval_dataset=dataset_samsum_pt["validation"])
5
```

```
1
2 trainer.train()
3
```

You're using a PegasusTokenizerFast tokenizer. Please note that with a fast tokeni
 [51/51 02:35, Epoch 0/1]

Step Training Loss Validation Loss

TrainOutput(global_step=51, training_loss=3.0761698928533816, metrics=
 {'train_runtime': 159.3528, 'train_samples_per_second': 5.14

▼ Evaluating the Model

The model is evaluated using the ROUGE metric, which is a set of metrics used to evaluate automatic summarization and machine translation. The `load_metric` function from the `datasets` library is used to load the ROUGE metric.

The evaluation process involves the following steps:

1. **Batching the Data:** The test data is divided into smaller batches using the `generate_batch_sized_chunks` function. This function takes a list of elements and a batch size as input and yields successive batch-sized chunks from the list of elements.
2. **Generating Summaries:** For each batch of articles, the model generates a batch of summaries. The `generate` method of the model is used for this purpose. The inputs to the model are tokenized versions of the articles. The `generate` method also takes a few additional parameters such as `length_penalty`, `num_beams`, and `max_length` to control the generation process.
3. **Decoding the Summaries:** The generated summaries are then decoded using the tokenizer. The `decode` method of the tokenizer is used for this purpose. The `skip_special_tokens` and `clean_up_tokenization_spaces` parameters are set to `True` to remove any special tokens and clean up the tokenization spaces.
4. **Calculating the Metric:** The decoded summaries and the target summaries are then added to the metric using the `add_batch` method. Finally, the metric is computed using the `compute` method.

Here is the code for evaluating the model:

```
1 # Evaluation
2
3 def generate_batch_sized_chunks(list_of_elements, batch_size):
4     """split the dataset into smaller batches that we can process simultaneously
5     Yield successive batch-sized chunks from list_of_elements."""
6     for i in range(0, len(list_of_elements), batch_size):
7         yield list_of_elements[i : i + batch_size]
8
9
10
11 def calculate_metric_on_test_ds(dataset, metric, model, tokenizer,
12                                batch_size=16, device=device,
13                                column_text="article",
14                                column_summary="highlights"):
15     article_batches = list(generate_batch_sized_chunks(dataset[column_text], batch_size)
16     target_batches = list(generate_batch_sized_chunks(dataset[column_summary], batch_si
17
18     for article_batch, target_batch in tqdm(
19         zip(article_batches, target_batches), total=len(article_batches)):
20
```

```

21     inputs = tokenizer(article_batch, max_length=1024, truncation=True,
22                          padding="max_length", return_tensors="pt")
23
24     summaries = model.generate(input_ids=inputs["input_ids"].to(device),
25                               attention_mask=inputs["attention_mask"].to(device),
26                               length_penalty=0.8, num_beams=8, max_length=128)
27     ''' parameter for length penalty ensures that the model does not generate seque
28
29     # Finally, we decode the generated texts,
30     # replace the token, and add the decoded texts with the references to the metr
31     decoded_summaries = [tokenizer.decode(s, skip_special_tokens=True,
32                                         clean_up_tokenization_spaces=True)
33                          for s in summaries]
34
35     decoded_summaries = [d.replace("", " ") for d in decoded_summaries]
36
37
38     metric.add_batch(predictions=decoded_summaries, references=target_batch)
39
40     # Finally compute and return the ROUGE scores.
41     score = metric.compute()
42     return score

```

```

1 rouge_names = ["rouge1", "rouge2", "rougeL", "rougeLsum"]
2 rouge_metric = load_metric('rouge')

```

Downloading builder script:

5.65k/? [00:00<00:00, 137kB/s]

▼ Calculating the Score and Saving the Model

The `calculate_metric_on_test_ds` function is used to calculate the ROUGE score on the test dataset. The function takes the test dataset, the ROUGE metric, the model, the tokenizer, the batch size, and the column names for the dialogue and summary as input. The function returns a dictionary with the ROUGE scores.

The ROUGE scores are then converted into a dictionary and displayed in a pandas DataFrame for easy visualization. The keys of the dictionary are the names of the ROUGE metrics, and the values are the corresponding scores.

Finally, the trained model is saved using the `save_pretrained` method of the model. The model is saved in a directory named "pegasus-samsum-model".

Here is the code for calculating the score and saving the model:

```

1
2 score = calculate_metric_on_test_ds(
3     dataset_samsum['test'][0:10], rouge_metric, trainer.model, tokenizer, batch_size =
4 )
5
6 rouge_dict = dict((rn, score[rn].mid.fmeasure ) for rn in rouge_names )

```

```
7
8 pd.DataFrame(rouge_dict, index = [f'pegasus'] )
```

```
100%|██████████| 5/5 [00:35<00:00, 7.05s/it]
      rouge1  rouge2  rougeL  rougeLsum
pegasus  0.017471    0.0  0.01512  0.014961
```

```
1 ## Save model
2 model_pegasus.save_pretrained("pegasus-samsum-model")
```

```
1
2 ## Save tokenizer
3 tokenizer.save_pretrained("tokenizer")

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

```
1
2 #Load
3
4 tokenizer = AutoTokenizer.from_pretrained("/content/tokenizer")
```

▼ Prediction

The model's performance is evaluated by generating a summary for a sample dialogue from the test dataset. The `pipeline` function from the transformers library is used to create a summarization pipeline with the trained model and tokenizer.

The `length_penalty` parameter in the `gen_kwargs` dictionary is used to control the length of the generated summary. A lower value results in shorter summaries, while a higher value results in longer summaries. The `num_beams` parameter is used for beam search, which is a search algorithm that considers multiple hypotheses at each step. The `max_length` parameter is used to limit the maximum length of the generated summary.

The dialogue, the reference summary, and the model's generated summary are then printed for comparison.

Here is the code for generating a summary with the model:

```
1
2 #Prediction
3
4 gen_kwargs = {"length_penalty": 0.8, "num_beams":8, "max_length": 128}
5
6
```

```

7
8 sample_text = dataset_samsum["test"][0]["dialogue"]
9
10 reference = dataset_samsum["test"][0]["summary"]
11
12 pipe = pipeline("summarization", model="pegasus-samsum-model",tokenizer=tokenizer)
13
14 ##
15 print("Dialogue:")
16 print(sample_text)
17
18
19 print("\nReference Summary:")
20 print(reference)
21
22
23 print("\nModel Summary:")
24 print(pipe(sample_text, **gen_kwargs)[0]["summary_text"])

```

Your max_length is set to 128, but your input_length is only 122. Since this is a sum
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:

Amanda: Ask Larry Amanda: He called her last time we were at the park together .<n>Ha

▼ FastAPI

This script is used to create a FastAPI application that serves as a web server for the text summarization model. The application provides two endpoints: one for training the model and another for generating summaries.

Here is a brief explanation of the code:

1. The script begins by importing the necessary modules and initializing a FastAPI application.

2. The `@app.get("/")` decorator creates a root endpoint that redirects users to the API documentation.
3. The `@app.get("/train")` decorator creates an endpoint that trains the model when accessed. The training process is initiated by running the `main.py` script using the `os.system` function. If the training is successful, the endpoint returns a success message. If an error occurs during training, the endpoint returns an error message.
4. The `@app.post("/predict")` decorator creates an endpoint that generates a summary for a given text. The endpoint receives the text as a POST request, creates an instance of the `PredictionPipeline` class, and calls its `predict` method to generate the summary. If an error occurs during prediction, the endpoint raises an exception.
5. The `if __name__=="__main__":` block runs the application on a local server at port 8080 when the script is run directly.

This FastAPI application provides a simple and efficient way to train the model and generate summaries using HTTP requests. It can be easily integrated into other applications or services.

```
1 from fastapi import FastAPI
2 import uvicorn
3 import sys
4 import os
5 from fastapi.templating import Jinja2Templates
6 from starlette.responses import RedirectResponse
7 from fastapi.responses import Response
8 from textSummarization.pipeline.prediction import PredictionPipeline
9
10
11 text:str = "What is Text Summarization?"
12
13 app = FastAPI()
14
15 @app.get("/", tags=["authentication"])
16 async def index():
17     return RedirectResponse(url="/docs")
18
19
20 @app.get("/train")
21 async def training():
22     try:
23         os.system("python main.py")
24         return Response("Training successful !!")
25
26     except Exception as e:
27         return Response(f"Error Occurred! {e}")
28
29
30 @app.post("/predict")
31 async def predict_route(text):
32     try:
```

```
33
34     obj = PredictionPipeline()
35     text = obj.predict(text)
36     return text
37 except Exception as e:
38     raise e
39
40
41 if __name__=="__main__":
42     uvicorn.run(app, host="0.0.0.0", port=8080)
```

FastAPI 0.1.0 **QA59**
/openapi.json

authentication ^

GET

/ Index

default ^

GET

/train Training

POST

/predict Predict Route

Schemas ^

HTTPValidationError