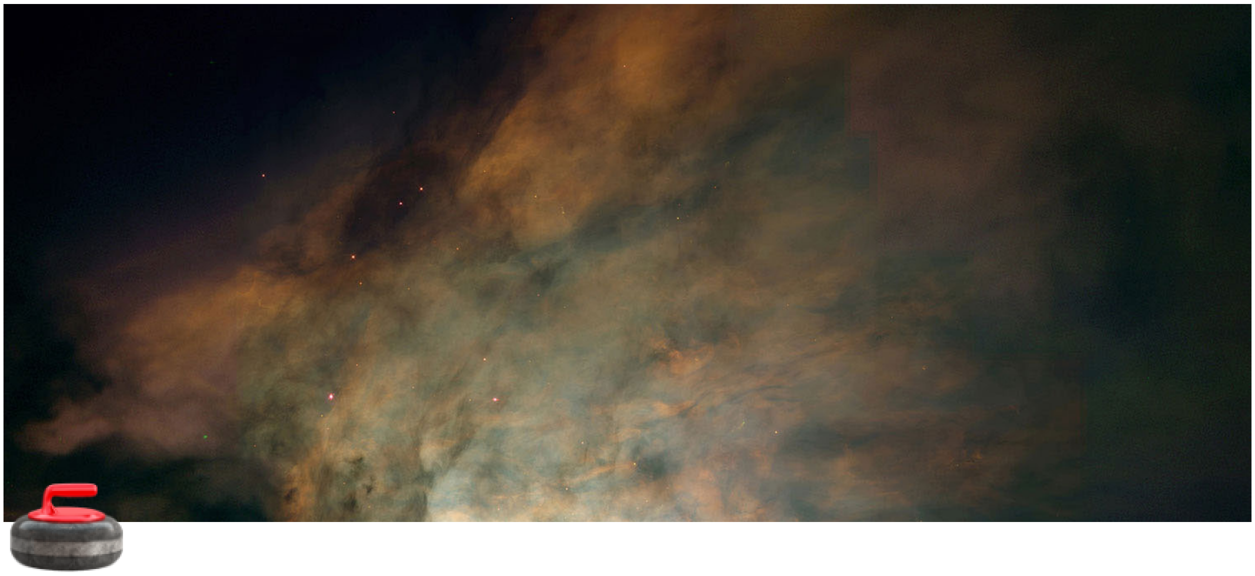


LLM :

Large language models (LLMs) are called “large” because they are **pre-trained** with a **large number of parameters** (100M+) on **large corpora of text** to process/understand and generate natural language text for a wide variety of NLP tasks.

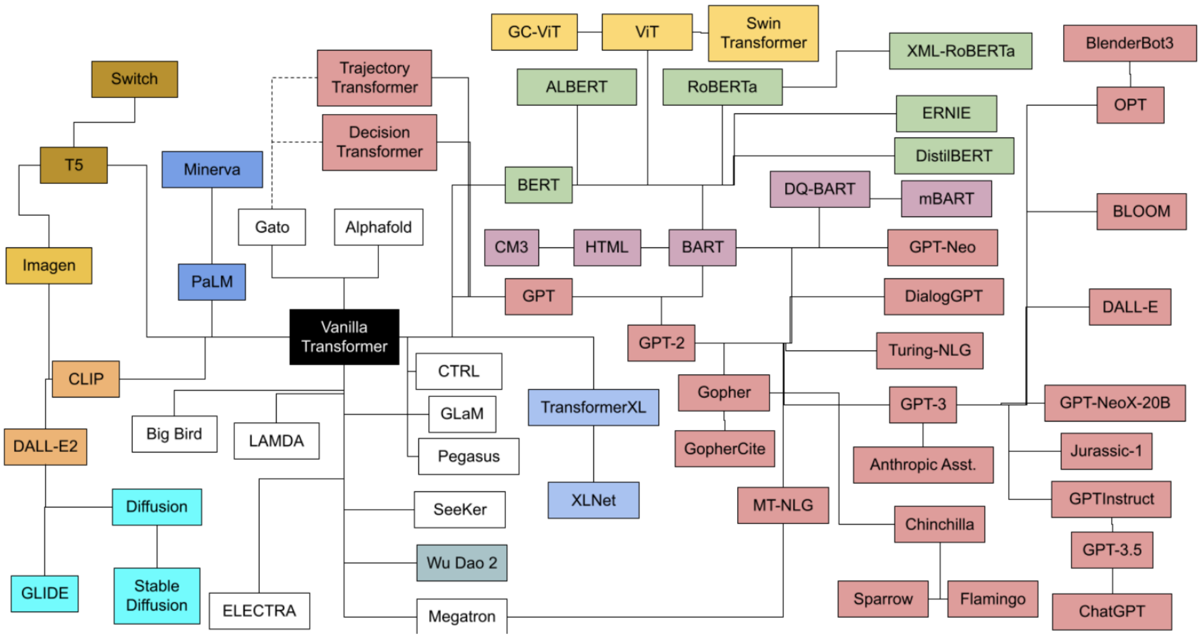
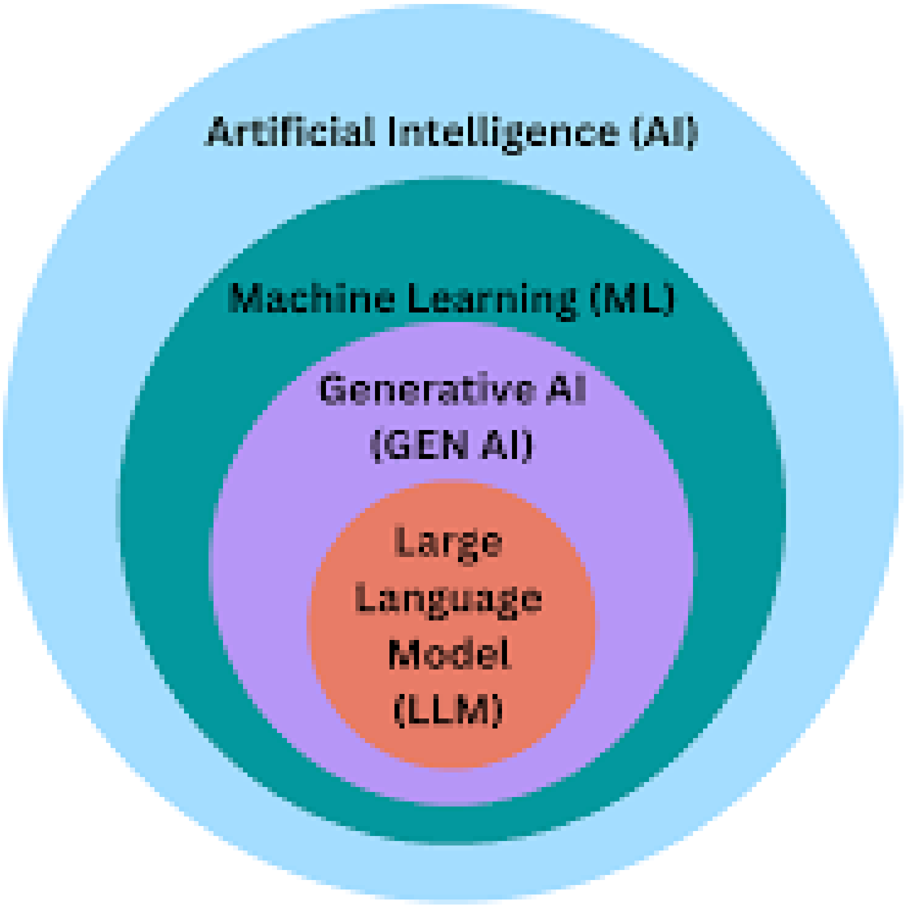


Figure 6: Transformers Family Tree

Large language models (LLMs) are advanced artificial intelligence systems designed to understand and generate human language. They utilize deep learning techniques and are trained on extensive datasets, often comprising billions of words, to perform various natural language processing (NLP) tasks such as text generation, translation, summarization, and question answering

Large Language Models (LLMs) are categorized based on their architecture, tasks, modalities, and access type (open-source or closed-source). Let's explore these categories in details:



Usability of LLMs (Large Language Models):

1. General-Purpose LLMs

General-purpose LLMs are designed to handle a wide range of tasks without domain-specific tuning. These models are trained on large corpora from the general web and are intended to be flexible for multiple applications like text generation, question answering, and summarization.

Examples:

- **GPT-3**: OpenAI's model with 175 billion parameters. It's capable of generating coherent text, answering questions, summarizing, translating, etc.
- **BERT**: Developed by Google, BERT (Bidirectional Encoder Representations from Transformers) is commonly used for downstream tasks like classification and question answering.

Use Case: These models are versatile and can be applied in many different fields, though they may not perform as well as domain-specific models for specialized tasks.

2. Domain-Specific LLMs

Domain-specific LLMs are trained on specialized text, such as biomedical, legal, or scientific data. They outperform general-purpose LLMs in their respective domains due to the specialized training.

Examples:

- **BioBERT**: A variant of BERT trained on biomedical literature for tasks like medical text mining.
- **SciBERT**: Optimized for scientific texts and used for tasks like scientific information extraction and research paper analysis.

Use Case: These models excel in specialized fields (e.g., medicine, law, science) where understanding domain-specific jargon and nuances is critical.

3. Multilingual LLMs

Multilingual LLMs are designed to support multiple languages, making them valuable for tasks like cross-lingual translation, multilingual sentiment analysis, and information retrieval across languages.

Examples:

- **XLM (Cross-lingual Language Model)**: Supports over 100 languages and is useful for cross-lingual tasks such as translation and alignment.

- **Multilingual BERT**: Trained on 104 languages, it can perform various NLP tasks across those languages, such as named entity recognition or text classification.

Use Case: These models are ideal for handling multilingual data and cross-lingual tasks like translation and entity recognition.

4. Few-Shot LLMs

Few-shot LLMs are designed to perform well with minimal fine-tuning, making them highly efficient for tasks with limited labeled data.

Examples:

- **GPT-3**: Known for its impressive few-shot learning capabilities.
- **T5 (Text-to-Text Transfer Transformer)**: Can achieve strong performance on tasks like summarization and translation with minimal data (as few as 10 examples).

Use Case: Few-shot models are useful when labeled data is scarce or expensive to obtain, allowing the model to generalize quickly from a few examples.

5. Task-Specific LLMs

Task-specific LLMs are tailored for particular NLP tasks, such as summarization, translation, or question answering. These models are fine-tuned to excel in their specific task, making them highly efficient.

Examples:

- **BART**: Facebook's model designed for text generation tasks such as summarization, translation, and question generation.
- **ALBERT**: A task-specific variant of BERT optimized for tasks like sentence classification and question answering.

Use Case: Task-specific models are perfect for scenarios where the task is well-defined, and task-specific optimization is beneficial, such as document summarization or machine translation.

Architectural Categories of LLMs

1. Autoregressive Models

Autoregressive models generate sequences one word at a time, predicting the next word based on previous ones. This makes them excellent for tasks that require text generation, such as dialogue systems or story generation.

Examples:

- **GPT-3 / GPT-4**: Uses an autoregressive approach to generate coherent text by predicting the next word.
- **Claude**: A conversational autoregressive model developed by Anthropic.

Use Case: Autoregressive models are predominantly used in language generation tasks where generating fluent and coherent text is important.

2. Autoencoding Models

Autoencoding models focus on understanding and reconstructing text. They are bidirectional, meaning they look at the entire context of a sequence (both left and right sides) to generate better representations, making them suitable for tasks like classification, sentence encoding, and question answering.

Examples:

- **BERT**: Google's bidirectional model trained on masked language modeling (MLM), allowing it to understand context better.
- **RoBERTa**: A robust version of BERT designed for better training efficiency.

Use Case: Autoencoding models are mainly used for understanding tasks (text classification, question answering) rather than text generation.

3. Encoder-Decoder Models

Encoder-decoder models involve two components: an encoder that processes the input and a decoder that generates the output. These models are highly effective for tasks like machine translation and summarization, where input-output mapping is necessary.

Examples:

- **T5**: Google's text-to-text model, capable of transforming various NLP tasks into a text generation framework.
- **BART**: Combines bidirectional encoding with autoregressive decoding, ideal for summarization and text generation tasks.

Use Case: Encoder-decoder models are best suited for tasks involving complex input-output mappings, like translation or summarization.

4. Multimodal Models

Multimodal models handle multiple types of data (text, images, audio) simultaneously. They are designed to integrate information from different data modalities to tackle complex tasks that go beyond just text.

Examples:

- **DALL-E**: Generates images from textual descriptions, combining vision and language.
- **CLIP**: Combines language and image understanding for tasks such as image recognition and captioning.

Use Case: Multimodal models are essential for tasks involving multiple types of data, such as text-to-image generation or vision-language understanding.

Access Categories: Open-Source vs. Closed-Source LLMs

1. Open-Source LLMs

Open-source LLMs are freely available for use and modification. They enable researchers and developers to explore, fine-tune, or build upon existing models for various applications.

Examples:

- **GPT-Neo/GPT-J**: Open-source alternatives to GPT-3, developed by EleutherAI.

- **T5**: Google's model is open-source and can be used for fine-tuning.
- **BERT/RoBERTa**: Both are available for public use and fine-tuning.

Use Case: Open-source models are beneficial for academic research, smaller companies, or individuals looking to experiment with large-scale models without incurring high costs.

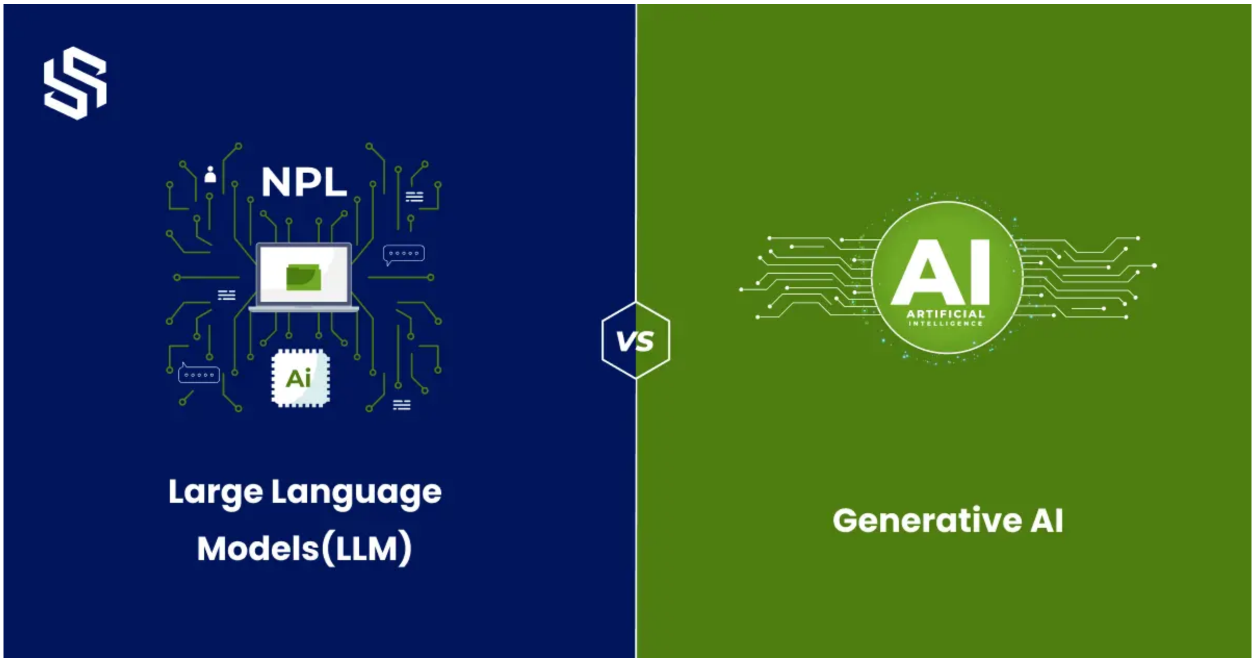
2. Closed-Source LLMs

Closed-source LLMs are proprietary and typically provided by companies through APIs or cloud services. Users cannot access or modify the underlying models, but they benefit from scalable, high-performance systems without requiring large computational resources.

Examples:

- **GPT-3 / GPT-4**: Available through OpenAI's API but not open-source.
- **Claude**: Closed-source model developed by Anthropic for conversational tasks.

Use Case: Closed-source models are ideal for organizations looking to integrate LLMs into production systems without the complexity of training or managing large models.



Generative AI vs. Large Language Models (LLMs):

Generative AI: The Power of Creation

Generative AI is a broad category focused on creating new content across multiple formats such

- **Text**: Writing paragraphs, poems, code, and more.
- **Images**: Creating or modifying artistic and realistic images.
- **Audio**: Generating music, sound effects, or speech.
- **Code**: Writing code in various programming languages.

Applications:

- Drug discovery, material science, creative design, music composition, etc.

Large Language Models (LLMs): Masters of Text

LLMs are a subset of generative AI focused on generating and understanding text. These models are trained on massive datasets to excel at:

- **Natural Language Understanding (NLU)**: Extracting meaning, identifying sentiment, and understanding context.
- **Text Generation**: Writing creative content like emails, letters, poems, or even functional code.
- **Machine Translation**: Converting text between languages.
- **Text Summarization**: Condensing long texts into shorter summaries.
- **Question Answering**: Answering queries with relevant and accurate information.

Key Differences Between Generative AI and LLMs:

- **Content Scope**:
  - **Generative AI**: Works across multiple formats (text, images, audio, code).
  - **LLMs**: Specialize in text processing and generation.
- **Techniques**:
  - **Generative AI**: Employs GANs, VAEs, and other models for generating diverse content types.
  - **LLMs**: Use transformer-based architectures specifically optimized for textual data.
- **Data Requirements**:
  - **Generative AI**: Requires large datasets specific to the content type (image datasets for image generation).
  - **LLMs**: Require massive amounts of text data for training to handle tasks like translation and summarization.

Applications of LLMs:

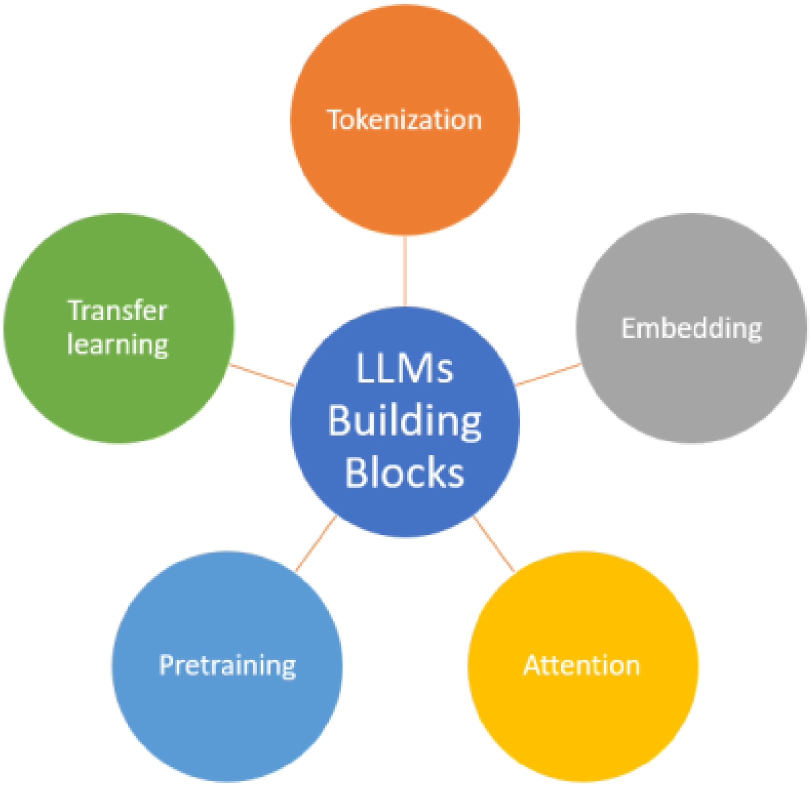
- **Chatbots**: Powering intelligent virtual assistants for customer support.
- **Content Creation**: Assisting writers and researchers with idea generation and drafting.
- **Education**: Developing personalized learning experiences and tutoring systems.
- **Code Generation**: Automating repetitive coding tasks.

Applications of Generative AI:

- **Drug Discovery**: Designing new molecules for pharmaceuticals.
- **Creative Design**: Generating visuals, marketing materials, and product designs.
- **Music Composition**: Creating unique musical pieces in various styles.

How does LLM work.

Large Language Models (LLMs) are composed of several key building blocks that enable them to efficiently process and understand natural language data.



- **Tokenization**: Converts text into individual words or tokens that models can process, using methods like Byte Pair Encoding (BPE) or WordPiece to handle both common and rare words efficiently.
- **Embedding**: Transforms tokens into vector representations in high-dimensional space, capturing the semantic meaning and relationships between words.
- **Attention**: Self-attention in transformers allows the model to focus on relevant words in a sequence, improving its ability to capture long-range dependencies in language.
- **Pre-training**: Initial training on large datasets to learn language patterns before fine-tuning for specific tasks, reducing labeled data needs.
- **Transfer Learning**: Adapts a pre-trained model to new tasks by fine-tuning on smaller, task-specific datasets, leveraging prior knowledge to improve performance.

LLMs Evolution :

1. Increasing Parameter Size

One of the most notable trends in LLM development is the dramatic increase in the number of parameters.

- **Example**: GPT-4 has approximately **1.8 trillion parameters**, which is about ten times more than its predecessor, GPT-3, which has 175 billion parameters. This increase allows GPT-4 to generate more nuanced and contextually relevant text compared to earlier models.

Impact

Larger models can capture complex language patterns and provide more accurate responses across diverse tasks, such as text generation, translation, and summarization.

2. Enhanced Model Architectures

The architecture of LLMs has also evolved to improve efficiency and performance.

- **Example**: The introduction of **Mixture-of-Experts (MoE)** architectures allows models like GPT-4 to activate only a subset of parameters during inference, significantly reducing computational costs while maintaining high performance. This approach enables the model to scale up without a proportional increase in resource requirements.

Impact

Such architectures allow for larger models that can operate efficiently on standard hardware, making advanced capabilities more accessible.

3. Better Training Techniques

Training methods have seen improvements that enhance model performance without solely relying on increased size.

- **Example**: Techniques like **reinforcement learning from human feedback (RLHF)** have been employed in models like ChatGPT to refine responses based on user interactions and preferences, leading to more human-like conversations.

Impact

These methods improve the model's ability to follow instructions and generate contextually appropriate responses, enhancing user experience.

4. Specialized Models

With the rise of LLMs, there has been a trend towards developing specialized models that cater to specific applications.

- **Example**: **Gemini Nano**, with 1.8 billion parameters, is designed for efficient performance on devices with limited resources, such as smartphones. It excels at tasks like summarizing text and suggesting replies while being less resource-intensive than larger models.

Impact

Specialized models demonstrate that efficiency can be prioritized without sacrificing performance, making advanced AI capabilities available on a wider range of devices.

- **Pre-2017**: Early models were small, rule-based, and statistical (n-grams, RNNs, word2vec). They had limited capabilities.
- **2017-2018**: Introduction of Transformers (like BERT, GPT-1) brought long-range context understanding, enabling more powerful models.
- **2019-2020**: Models scaled massively, with GPT-2 and GPT-3 (up to 175B parameters) showing strong performance and few-shot learning.

- **2021-2022**: Focus shifted to efficiency and usability, leading to smaller models (like GPT-Neo) and easier access (via Hugging Face, etc.).
- **2023-Present**: Open-source LLMs, specialized fine-tuning, and MLOps tools made deployment easier. Focus is now on smaller, multi-modal, and more ethical models.

How GPT Trained :

Training Stages of ChatGPT:

Stage 1 — Generative Pre-Training:

- **Objective:** Train the model on diverse internet text (websites, books, articles) for text generation tasks.
- **Key Tasks:** Language modeling, summarization, translation, sentiment analysis, text completion.
- **Challenge:** Misalignment between user expectations and the model's capabilities. The model is versatile but not specialized for conversational tasks.

Stage 2 — Supervised Fine-Tuning (SFT):

- **Objective:** Fine-tune the model for conversation tasks using carefully curated training data.
- **Process:**
  1. Create conversational datasets where one human simulates chatbot responses.
  2. Train the model to predict the ideal next response using supervised learning (Stochastic Gradient Descent).

- **Challenge: Distributional Shift**—the model is trained on specific tasks but struggles with broader conversations.

Stage 3 — Reinforcement Learning through Human Feedback (RLHF):

- **Objective:** Refine the model using human feedback to make decisions that align with user preferences.
- **Process:**
  1. Human agents rank different model responses.
  2. Train a reward model that scores responses based on rankings.
  3. Fine-tune ChatGPT using Proximal Policy Optimization (PPO) to improve conversational quality.
- **Challenge: Goodhart's Law**—the model can over-optimize the reward, leading to unintended behavior.

Tokenization:

Purpose of Tokenization:

- **Why Tokenization is Necessary**: Tokenization is the process of converting raw text into a format that can be processed by machine learning models. Models typically work with numerical data, so tokenization translates text into tokens (words, subwords, or characters) and then into numerical IDs. This conversion is essential because models cannot understand raw text and need standardized input for training and prediction.

How to Choose a Tokenizer:

- **Match Tokenizer to Model**: Always use the tokenizer that matches the model architecture. For example, use `BertTokenizer` with BERT models and `GPT2Tokenizer` with GPT-2 models to ensure compatibility in tokenization and model input.

The **encoded output** returned by the tokenizer in Hugging Face includes several components, such as `input_ids`, `attention_mask`, and `token_type_ids`. These components are crucial when feeding the inputs into models like BERT, GPT-2, RoBERTa, and others. Let's break down each component:

1. input_ids

- **Definition**: These are the numerical IDs representing each token in the input text. Each model has its own vocabulary, and each token from the text is mapped to a unique ID in that vocabulary.
- **Example**:
  - If a token like "Hello" is mapped to the ID `7592` in the BERT tokenizer, the `input_ids` will include this ID.
- **Use in Models**: `input_ids` represent the input data that the model processes.

2. attention_mask

- **Definition**: The attention mask indicates which tokens should be attended to by the model and which should be ignored (usually padding tokens). It contains `1`s for actual tokens and `0`s for padding tokens.
- **Why Important?**: When you pad sequences to make them the same length, padding tokens are not meaningful. The attention mask ensures that the model doesn't treat these padding tokens as relevant information.
- **Example**:
  - `attention_mask = [1, 1, 1, 1, 1, 0, 0]` means the first five tokens are real tokens, and the last two are padding tokens that should be ignored.

3. token_type_ids

- **Definition**: These indicate different segments of the input, mainly useful in models like BERT that are used for tasks involving multiple sentences (like question-answering). It helps the model distinguish between two different sentences or parts of the input.
    - **Segment 0**: Usually represents the first sentence or segment.
    - **Segment 1**: Usually represents the second sentence or segment in tasks like sentence-pair classification (e.g., in question-answering).
- **Why Important?**: In tasks like sentence pair classification, the model needs to know which tokens belong to which sentence.
- **Example**:
    - `token_type_ids = [0, 0, 0, 1, 1]` means the first three tokens belong to the first sentence, and the last two tokens belong to the second sentence.

Example Breakdown

Let's revisit this encoded output:

```
Encoded Inputs:
{'input_ids': tensor([[ 101, 7592, 1010, 2129, 2024, 2017, 10
29,  102,    0],
                      [ 101, 1045, 2572, 2986, 1010, 4067, 20
17, 1012,  102]]),
 'token_type_ids': tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0],
                           [0, 0, 0, 0, 0, 0, 0, 0, 0]]),
 'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 0],
                           [1, 1, 1, 1, 1, 1, 1, 1, 1]])}
```

1. **input_ids**:
    - `[ 101, 7592, 1010, 2129, 2024, 2017, 1029, 102, 0]`
        - `101`: `[CLS]` token (special token added by the tokenizer).

    - `7592`: ID for "Hello".
    - `1010`: ID for ",".
    - `2129`: ID for "how".
    - `2024`: ID for "are".
    - `2017`: ID for "you".
    - `1029`: ID for "?".
    - `102`: `[SEP]` token (used to separate sentences).
    - `0`: Padding (added to make the sequence of fixed length).

2. **token_type_ids**:
    - `[0, 0, 0, 0, 0, 0, 0, 0, 0]`
        - All tokens belong to a single segment (sentence), hence all values are `0`.
3. **attention_mask**:
    - `[1, 1, 1, 1, 1, 1, 1, 1, 0]`
        - The first 8 values (including the special tokens `[CLS]` and `[SEP]`) are real tokens, and the `0` corresponds to the padding token, which is ignored by the model during attention calculations.

Why Are These Important?

- **input_ids**: They are the main input data for the model.
- **attention_mask**: Ensures the model focuses only on the relevant tokens and ignores padding.
- **token_type_ids**: Essential for tasks that involve multiple segments, like sentence-pair classification or question answering.

1. Encoding: Converting Text to Tokens and Token IDs

Encoding involves converting a piece of text into a sequence of tokens and then mapping those tokens to their corresponding IDs (input for the model).

Example Code for Encoding:

```
python
Copy code
from transformers import BertTokenizer

# Load the pre-trained BERT tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

# Text to encode
text = "Hello, how are you?"

# Encoding the text into tokens and token IDs
encoded = tokenizer.encode_plus(text, add_special_tokens=True, return_tensors='pt')

# Print encoded result
print("Encoded Input IDs:", encoded['input_ids'])
print("Attention Mask:", encoded['attention_mask'])
```

Explanation:

- `encode_plus`: This method performs both tokenization and encoding into token IDs. It can also add special tokens like `[CLS]` and `[SEP]`.
- `return_tensors='pt'`: Returns the results as PyTorch tensors. You can also use `'tf'` for TensorFlow tensors.

Output Example:

```
plaintext
Copy code
```

```
Encoded Input IDs: tensor([[  101,  7592,  1010,  2129,  202
4,  2017,  1029,   102]])
Attention Mask: tensor([[1, 1, 1, 1, 1, 1, 1, 1]])
```

- `101`: `[CLS]` token
- `7592`: "Hello"
- `1010`: ","
- `2129`: "how"
- `102`: `[SEP]` token

2. Decoding: Converting Token IDs Back to Text

Decoding involves converting the token IDs back into a readable string format.

Example Code for Decoding:

```
python
Copy code
# Token IDs (usually obtained after tokenization/encoding or
model output)
token_ids = [101, 7592, 1010, 2129, 2024, 2017, 1029, 102]

# Decoding the token IDs back to text
decoded_text = tokenizer.decode(token_ids)

# Print the decoded text
print("Decoded Text:", decoded_text)
```

Output Example:

```
plaintext
Copy code
```

```
Decoded Text: [CLS] hello, how are you? [SEP]
```

1. [CLS] Token (Classification Token)

- **Purpose**:
  - The `[CLS]` token is used as the first token in the input sequence. It serves as a placeholder for the aggregate representation of the entire sequence.
  - In tasks like sentence classification or question-answering, the model uses the hidden state of this token to make predictions.
- **Why Add It?**: The `[CLS]` token allows models to perform tasks like sequence classification. The final hidden state corresponding to this token is used as the representation of the entire input sequence.

2. [SEP] Token (Separator Token)

- **Purpose**:
  - The `[SEP]` token is used to separate different segments or sentences in a sequence.
  - For tasks that require multiple inputs (e.g., question-answering or sentence pair classification), this token helps the model distinguish between different parts of the input.
- **Why Add It?**: Models like BERT are trained to understand that different segments are separated by `[SEP]`, so it's crucial for sentence-pair or multi-segment tasks.

3. [PAD] Token (Padding Token)

- **Purpose**:
    - The `[PAD]` token is used to ensure that all input sequences are of the same length by padding shorter sequences with this token.
    - It ensures that the input has a uniform shape, which is required for batch processing in models.
- **Why Add It?**: Models expect input sequences of the same length in batch processing, so padding is used to make sure shorter sequences match the length of the longest sequence in the batch.

Code Example: Using Special Tokens with Hugging Face Tokenizer

Here's an example that demonstrates the usage of special tokens with the BERT tokenizer:

```
from transformers import BertTokenizer

# Load the pre-trained BERT tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

# Example input sentences
sentence1 = "Hello, how are you?"
sentence2 = "I'm fine, thank you."

# Tokenize the sentences with special tokens
encoded = tokenizer.encode_plus(
    sentence1,
    sentence2,                # Provide the second sentence
    add_special_tokens=True,   # Add [CLS], [SEP] tokens
    padding='max_length',      # Pad to a fixed length
    max_length=12,             # Max length of the sequence
    truncation=True,           # Truncate if input exceeds max length
    return_tensors='pt'        # Return PyTorch tensors
)

# Print the encoded outputs
print("Encoded Input IDs: ", encoded['input_ids'])
print("Token Type IDs: ", encoded['token_type_ids'])
print("Attention Mask: ", encoded['attention_mask'])

# Decode the input IDs back to the original sentences
decoded = tokenizer.decode(encoded['input_ids'][0], skip_spec
ial_tokens=False)
print("Decoded Sentence: ", decoded)
```

Example Output

The `encoded` output will have the following structure:

```
plaintext
Copy code
Encoded Input IDs:  tensor([[ 101, 7592, 1010, 2129, 2024, 20
17, 1029,  102, 1045, 1005, 1049, 102]])
Token Type IDs:     tensor([[0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1,
1]])
Attention Mask:     tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1]])
Decoded Sentence:   [CLS] Hello, how are you? [SEP] I'm fine,
thank you. [SEP]
```

Explanation of Output:

1. **input_ids**:

    - `101`: `[CLS]` token
    - `7592`: "Hello"

- `1010`: ","
- `2129`: "how"
- `2024`: "are"
- `2017`: "you"
- `1029`: "?"
- `102`: `[SEP]` token
- `1045`: "I"
- `1005`: "'"
- `1049`: "m"
- `102`: `[SEP]` token

2. **token_type_ids**:
   - The first part of the sequence (sentence 1) is marked with `0`s.
   - The second part of the sequence (sentence 2) is marked with `1`s.
3. **attention_mask**:
   - Every token (including `[CLS]`, `[SEP]`) is marked with `1` because they are actual tokens that need to be attended to (no padding here).

Why Add These Special Tokens?

- **[CLS]**: Helps the model understand that the sentence starts here and is used for classification.
- **[SEP]**: Helps the model know where one segment ends and the next begins.
- **[PAD]**: Ensures uniform input length across batches, which is important for batch processing.

Padding

What is Padding?

- Padding is the process of adding extra tokens (usually the `[PAD]` token) to shorter sequences so that they reach a fixed length. This ensures that all sequences in a batch have the same length, which is necessary for efficient batch processing in models.

Why Do We Use Padding?

- **Uniform Input Size**: Deep learning models, especially transformers, require inputs to have a uniform shape (same length) when processing in batches. Padding ensures that shorter sequences match the length of the longest sequence in the batch.
- **Efficiency**: Models can handle multiple sequences at once (batch processing), which is faster than processing each sequence individually. Padding helps align these sequences in terms of length.

Example Code for Padding:

```
from transformers import BertTokenizer

# Load BERT tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

# Example input sentence
sentence = "Hello"

# Tokenizing with padding to a fixed max length of 10
encoded = tokenizer(sentence, padding='max_length', max_length=10, return_tensors='pt')

# Print padded input IDs
print("Padded Input IDs:", encoded['input_ids'])
```

Output Explanation:

```
plaintext
Copy code
Padded Input IDs: tensor([[ 101, 7592,  102,    0,    0,
0,    0,    0,    0,    0]])
```

- `101`: `[CLS]` token
- `7592`: "Hello"
- `102`: `[SEP]` token
- `0`: `[PAD]` tokens added to make the sequence length 10.

Here, the sequence "Hello" has been padded with `[PAD]` tokens (represented by `0`) to reach a total length of 10 tokens.

2. Truncation

What is Truncation?

- Truncation is the process of shortening sequences that exceed a certain maximum length by removing tokens from the end. This ensures that no sequence is longer than a specified maximum length, preventing model input from becoming too large.

Why Do We Use Truncation?

- **Fixed Input Size**: Some models have a limitation on the maximum sequence length they can process (e.g., BERT typically handles up to 512 tokens). If a sequence is too long, it needs to be truncated to fit within this limit.
- **Memory Efficiency**: Reducing the sequence length via truncation helps conserve memory and ensures faster processing.

Example Code for Truncation:

```
long_sentence = "This is a very long sentence"

# Tokenizing with truncation to a max length of 10
```

```
encoded = tokenizer(long_sentence, truncation=True, max_length=10, return_tensors='pt')

# Print truncated input IDs
print("Truncated Input IDs:", encoded['input_ids'])
```

Output Explanation:

```
plaintext
Copy code
Truncated Input IDs: tensor([[ 101, 2023, 2003, 1037, 2200, 2146, 6251,  102]])
```

- `101`: `[CLS]` token
- `2023`: "This"
- `2003`: "is"
- `1037`: "a"
- `2200`: "very"
- `2146`: "long"
- `6251`: "sentence"
- `102`: `[SEP]` token

Project text Summarization :

Model

- **Model:** `google/pegasus-cnn_dailymail`

Imports

- `from transformers import AutoModelForSeq2SeqLM, AutoTokenizer, pipeline`
- `import torch`
- `import pandas as pd`
- `from datasets import load_dataset`
- `import evaluate`
- `from transformers import DataCollatorForSeq2Seq, TrainingArguments, Trainer`
- `from tqdm import tqdm`

Dataset

- **Dataset:** `samsum` (loaded using `datasets.load_dataset("samsum")`)

Steps ...

1. **Setup Environment:**
    - Install necessary libraries.
    - Import required modules.
2. **Model and Tokenizer Initialization:**
    - Load pre-trained model and tokenizer.
    - Move model to the appropriate device (CPU/GPU).
3. **Data Preparation:**
    - Load the dataset.
    - Convert dataset examples to features using a custom function.
    - Map the function to the dataset.

4. **Training Configuration:**
    - Set up data collator for sequence-to-sequence tasks.
    - Define training arguments.
    - Initialize the `Trainer` with model, tokenizer, data collator, and datasets.
5. **Training:**
    - Train the model using the `Trainer`.
6. **Evaluation:**
    - Define functions to split data into batches and calculate metrics.
    - Calculate ROUGE scores on the test dataset.
7. **Save Model and Tokenizer:**
    - Save the trained model and tokenizer to specified directories.
8. **Load Model and Tokenizer:**
    - Reload the model and tokenizer from saved directories.
9. **Prediction:**
    - Generate summaries for sample texts using the trained model.
    - Print dialogue, reference summary, and model-generated summary.

- **`transformers[sentencepiece]`**: Provides tools for working with pre-trained models for NLP tasks. The `sentencepiece` extra is needed for models that use SentencePiece tokenization.
- **`datasets`**: Helps load and work with various datasets in NLP tasks efficiently.
- **`sacrebleu`**: Used for evaluating machine translation outputs by computing BLEU scores.

- `rouge_score`: Provides metrics for evaluating text summarization by calculating ROUGE scores.
- `py7zr`: Allows handling `.7z` archive files, which might be necessary for extracting datasets or models.

- `!pip install --upgrade accelerate`: Updates the `accelerate` library to the latest version. `accelerate` helps with optimizing model training across different hardware setups.
- `!pip uninstall -y transformers accelerate`: Uninstalls both `transformers` and `accelerate` libraries. This might be done to resolve conflicts or issues with the current versions.
- `!pip install transformers accelerate`: Reinstalls the `transformers` and `accelerate` libraries. This ensures you have the latest compatible versions after resolving previous conflicts.

- `AutoModelForSeq2SeqLM`: This is a class used to load a pre-trained sequence-to-sequence (seq2seq) model. These models are commonly used for tasks like translation, summarization, and other text generation tasks.
- `AutoTokenizer`: This is a class used to load a tokenizer that corresponds to the pre-trained model. Tokenizers are essential for converting raw text into the format required by the model (e.g., converting text into token IDs).

- **Function Definition**:

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

  - **Purpose**: Defines a function that processes a batch of examples to convert them into the format required by the model.

- **Tokenize Dialogue**:

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

  - **Purpose**: Tokenizes the `dialogue` text from the batch, setting a maximum length of 1024 tokens and truncating if necessary.

- **Tokenize Summary**:

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

  - **Purpose**: Tokenizes the `summary` text from the batch, setting a maximum length of 128 tokens and truncating if necessary. The `with tokenizer.as_target_tokenizer()` context is used to apply target-specific settings if needed.

- **Return Tokenized Features**:

  ```
  return {
      'input_ids': input_encodings['input_ids'],
  ```

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

- **Purpose**: Returns a dictionary containing:
  - `input_ids`: Token IDs for the input (dialogue).
  - `attention_mask`: Mask to indicate which tokens are actual input and which are padding.
  - `labels`: Token IDs for the target (summary).

Purpose of `map`

1. **Transform Data**:
   - **Goal**: Convert raw data into a format suitable for model training or evaluation.
   - **Action**: The `map` function applies the `convert_examples_to_features` function to each example in the dataset.
2. **Batch Processing**:
   - **Goal**: Improve efficiency by processing data in batches.
   - **Action**: The `batched=True` argument allows processing multiple examples at once, which is faster than processing one example at a tim

Why Use Data Collator:

- **Efficiency**: Automates the process of preparing batches, making it easier to train the model efficiently.

Creating TrainingArguments Instance:

- **Parameters**:
    - `output_dir='pegasus-samsum'`: Specifies the directory where the model checkpoints and other outputs will be saved.
    - `num_train_epochs=1`: Sets the number of times the model will go through the entire training dataset (1 epoch).
    - `warmup_steps=500`: Defines the number of steps for the learning rate warmup, which helps stabilize training at the beginning.
    - `per_device_train_batch_size=1`: Sets the batch size for training on each device (e.g., GPU or CPU).
    - `per_device_eval_batch_size=1`: Sets the batch size for evaluation on each device.
    - `weight_decay=0.01`: Applies weight decay (regularization) to avoid overfitting by penalizing large weights.
    - `logging_steps=10`: Defines how often (in steps) training logs will be recorded.
    - `evaluation_strategy='steps'`: Specifies the strategy for evaluation. Here, it evaluates the model every specified number of steps.
    - `eval_steps=500`: Sets the number of steps between evaluations.
    - `save_steps=1e6`: Defines how often (in steps) the model checkpoints will be saved. `1e6` means a very large number, effectively saving infrequently.
    - `gradient_accumulation_steps=16`: Accumulates gradients over multiple steps before updating model parameters, which allows for a larger effective batch size without requiring more memory.

Steps and Their Purpose

1. **Splitting the Dataset into Batches**

```
def generate_batch_sized_chunks(list_of_elements, batch_si
ze):
    for i in range(0, len(list_of_elements), batch_size):
        yield list_of_elements[i: i + batch_size]
```

- **Purpose**: This helps handle large datasets that can't fit into memory all at once by processing them in manageable chunks.
- **Why**: This ensures efficient processing and avoids memory issues.

2. **Tokenizing Inputs and Targets**

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

- **Purpose**: Convert raw text into token IDs that the model can understand.
- **Why**: Models work with token IDs, not raw text, so this step is crucial for generating predictions.

3. **Generating Summaries**

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

- **Purpose**: Use the trained model to generate summaries based on the input text.
- **Why**: This step produces the model's output, which will be compared against reference summaries to evaluate performance.

4. **Decoding the Generated Summaries**

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

- **Purpose**: Convert token IDs back into human-readable text.
- **Why**: The ROUGE metric operates on text, not token IDs, so decoding is necessary for comparison.

5. **Replacing Special Tokens**

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

- **Purpose**: Clean up any special tokens or formatting issues.
- **Why**: Ensures the generated summaries are in a readable format.

6. **Adding Predictions and References to Metric**

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

- **Purpose**: Provide the metric with the model’s predictions and the reference summaries for comparison.

- **Why**: This is necessary for calculating the ROUGE score, which compares these pairs to evaluate the model's performance.

7. **Computing the ROUGE Score**

   ```
   score = metric.compute()
   ```

   - **Purpose**: Calculate the ROUGE score based on the provided predictions and references.
   - **Why**: This final step provides a quantitative measure of how well the generated summaries match the reference summaries.

ROUGE Metric Names

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

- **Purpose**: Define the specific ROUGE metrics you want to calculate.
- **Explanation**:
  - **"rouge1"**: Measures the overlap of unigrams (single words) between the generated and reference summaries.
  - **"rouge2"**: Measures the overlap of bigrams (two-word sequences) between the generated and reference summaries.
  - **"rougeL"**: Measures the longest common subsequence (LCS) between the generated and reference summaries, considering the order of words.

    - **"rougeLsum"**: Similar to ROUGE-L but applied to the entire summary.

Calculate Metrics

```
score = calculate_metric_on_test_ds(
    dataset_samsum['test'][0:10], rouge_metric, trainer.mode
l, tokenizer, batch_size=2, column_text='dialogue', column_su
mmary='summary'
)
```

- **Purpose**: Calculate ROUGE scores for a subset of the test dataset.
- **Explanation**:
    - `dataset_samsum['test'][0:10]`: Selects the first 10 samples from the test dataset.
    - `rouge_metric`: The ROUGE metric object used for calculating scores.
    - `trainer.model`: The model used to generate summaries.
    - `tokenizer`: The tokenizer for processing text.
    - `batch_size=2`: Specifies the batch size for processing data.
    - `column_text='dialogue'`: Indicates the column in the dataset containing the input text.
    - `column_summary='summary'`: Indicates the column in the dataset containing the reference summaries.
    - `calculate_metric_on_test_ds(...)`: Function call that computes ROUGE scores for the provided subset of the test dataset.

Update ROUGE Scores

```
rouge_dict = dict((rn, score[rn]) for rn in rouge_names)  # J
ust use score[rn]
```

- **Purpose**: Extract the ROUGE scores from the computed results.
- **Explanation**:
    - `score[rn]`: Retrieves the ROUGE score for each metric specified in `rouge_names`.
    - `rouge_dict`: A dictionary where each key is a ROUGE metric name and each value is the corresponding score.

Display Results

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

- **Purpose**: Create a DataFrame to display the ROUGE scores in a readable format.
- **Explanation**:
    - `pd.DataFrame(rouge_dict, index=[f'pegasus'])`: Converts the `rouge_dict` into a DataFrame with the index labeled `'pegasus'`. This allows you to neatly view the ROUGE scores for the model.

Save Model

```
python
Copy code
```

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

- **Purpose**: Saves the trained model's configuration and weights.
- **Explanation**: Stores the model's architecture and learned parameters in the `"pegasus-samsum-model"` directory for future use or further training.

Save Tokenizer

```
python
Copy code
tokenizer.save_pretrained("tokenizer")
```

- **Purpose**: Saves the tokenizer's configuration and vocabulary.
- **Explanation**: Stores tokenizer settings and vocabulary in the `"tokenizer"` directory, allowing you to reuse the tokenizer without reinitializing or retraining.

Loading the Tokenizer

```
python
Copy code
tokenizer = AutoTokenizer.from_pretrained("/content/tokenizer")
```

Purpose: To load the tokenizer that was previously saved.

Explanation:

- `AutoTokenizer.from_pretrained("/content/tokenizer")`: This command tells the tokenizer to fetch its configuration and vocabulary from the directory located at `/content/tokenizer`.

- `/content/tokenizer`: This is the directory where the tokenizer's settings and vocabulary were saved. By loading from this directory, you ensure that the tokenizer can process text data in the same way it did during training.

Summary :

Explanation:

- `gen_kwargs = {"length_penalty": 0.8, "num_beams": 8, "max_length": 128}`: Defines parameters for text generation:
    - `length_penalty`: Penalizes longer sequences to avoid overly lengthy summaries.
    - `num_beams`: Specifies the number of beams for beam search, which improves summary quality.
    - `max_length`: Sets the maximum length of the generated summary.
- `sample_text` and `reference`: Extract the dialogue and reference summary from the test dataset for evaluation.
- `pipe = pipeline("summarization", model="pegasus-samsum-model", tokenizer=tokenizer)`: Creates a summarization pipeline using the trained model and tokenizer.
- `print("Dialogue:")` and `print("Reference Summary:")`: Output the input dialogue and the reference summary for comparison.
- `print("\nModel Summary:")`: Generates and prints the model's summary for the input dialogue using the defined generation parameters.

chat compeletion Parameter:

Required Parameters

1. `model`:

   Specifies which model to use. For example:

   - `"gpt-4"`
   - `"gpt-3.5-turbo"`

2. `messages`:

   A list of message objects describing the conversation so far. The format is:

   ```
   json
   Copy code
   [
     {"role": "system", "content": "You are a helpful assistant."},
     {"role": "user", "content": "Tell me a joke."}
   ]
   ```

   - `role`: The role of the message sender. Can be one of:
     - `"system"`: Sets behavior or instructions for the assistant.
     - `"user"`: The actual user input or query.
     - `"assistant"`: Previous messages from the assistant.
   - `content`: The content of the message.

Optional Parameters

1. `temperature` (default: 1):

   Controls the randomness of the output.

   - Range: 0 to 2.

- Higher values like 1.5 will make the output more random, while lower values like 0.2 will make it more deterministic.

2. `max_tokens`:

   Sets the maximum number of tokens to generate in the response. The total token count (input + output) is limited by the model's maximum (e.g., 4096 tokens for `gpt-3.5-turbo` and 8192 or 32768 for `gpt-4`).

3. `top_p` (default: 1):

   Controls the cumulative probability threshold for sampling.

   - Similar to temperature, but instead of controlling randomness directly, it limits the pool of words.
   - Lower values keep the output focused, while higher values make it more diverse.

4. `n` (default: 1):

   Specifies how many completion choices to generate for each input message.

5. `stream` (default: `false`):

   If `true`, the response will be sent back incrementally in a stream, useful for real-time applications.

6. `stop`:

   Specifies one or more tokens that will stop the generation when encountered. You can pass:

   - A single string or a list of strings.
   - Example: `stop=["\n\n"]` to stop when two newlines are encountered.

7. `presence_penalty` (default: 0):

   A value between `-2.0` and `2.0` that encourages or discourages the model from introducing new topics or concepts.

   - Positive values make the model more likely to introduce new information.

8. `frequency_penalty` (default: 0):

A value between `-2.0` and `2.0` that penalizes repeating the same lines verbatim.

- Positive values decrease the likelihood of repeating the same information.

9. `logit_bias`:

   A map from token IDs to biases, modifying the likelihood of specific tokens appearing in the response.

   - Example: `{"50256": -100}` could make a token extremely unlikely to appear.

10. `user`:

    A string to identify the end-user for monitoring or tracking abuse, and to improve safety features.

Prompting :

1. Zero-Shot Learning

You give no example or context, and the model has to respond based on the given prompt.

Example Prompt:

"How can I effectively prepare for my board exams?"

2. One-Shot Learning

You provide a single example to guide the response.

Example Prompt:

"How can I effectively prepare for my board exams?

Example: 'You should create a study schedule and stick to it every day.'"

3. Few-Shot Learning

You provide a few examples to help the model understand the type of response you expect.

Example Prompt:

"How can I effectively prepare for my board exams?

Example 1: 'Break down the syllabus into smaller parts and cover one section each day.'

Example 2: 'Practice past papers regularly to get familiar with the exam format.'"

4. Chain-of-Thought Prompting

You guide the model to break down its reasoning step by step.

Example Prompt:

"What steps should I take to prepare for my board exams effectively? Think step by step."

Expected Response:

"First, review the syllabus and prioritize important subjects. Next, create a daily study plan. Then, set aside time for regular revision. Finally, solve past exam papers to assess your preparation."

5. Iterative Prompting

You refine the prompt based on feedback, improving the response over iterations.

Example Prompt (First Iteration):

"How can I prepare for my board exams?"

If the response is too general:

"Can you give me a detailed study plan with subject-specific tips for my board exams?"

6. Negative Prompting

You specify what you **don't** want in the response.

Example Prompt:

"How can I prepare for my board exams without feeling stressed?"

7. Hybrid Prompting

This combines multiple techniques, such as using few-shot learning and chain-of-thought.

Example Prompt:

"How can I study for my board exams effectively?

Example 1: 'Set achievable daily goals.'

Example 2: 'Use flashcards for quick revision.'

Now, explain the detailed steps I should follow to prepare efficiently."

8. Prompt Chaining

You break the task into multiple smaller prompts.

Example Prompt 1:

"How should I create a study schedule for my board exams?"

Example Prompt 2 (after the first prompt):

"How should I revise effectively for each subject in the schedule?"

Example Prompt 3 (after the second prompt):

"How can I test myself to know if I'm ready for the exam?"

1. Zero-Shot Learning

No examples are provided. The model responds based on the prompt.

```
python
Copy code
prompt = "How can I effectively prepare for my board exams?"
print(get_completion(prompt))
```

2. One-Shot Learning

One example is provided to guide the model's response.

```
python
Copy code
prompt = """
How can I effectively prepare for my board exams?
Example: 'Create a study schedule and stick to it every day.'
"""
print(get_completion(prompt))
```

3. Few-Shot Learning

A few examples are provided to help guide the model.

```
python
Copy code
prompt = """
How can I effectively prepare for my board exams?
Example 1: 'Divide the syllabus into smaller parts and focus
on one section at a time.'
Example 2: 'Review past papers to get familiar with the exam
pattern.'
"""
print(get_completion(prompt))
```

4. Chain-of-Thought Prompting

Guide the model to explain its reasoning step by step.

```
python
Copy code
prompt = """
What steps should I take to prepare for my board exams? Think
step by step.
```

```
"""
print(get_completion(prompt))
```

5. Iterative Prompting

Refine the initial prompt for more detail or accuracy.

```
python
Copy code
# First iteration
prompt = "How can I prepare for my board exams?"
response = get_completion(prompt)
print(response)

# Refine based on feedback
refined_prompt = "Can you give me a subject-specific study pl
an for board exams?"
print(get_completion(refined_prompt))
```

6. Negative Prompting

You specify what **not** to include in the response.

```
python
Copy code
prompt = """
How can I prepare for my board exams without feeling stressed
or overwhelmed?
"""
print(get_completion(prompt))
```

7. Hybrid Prompting

Combines few-shot learning and chain-of-thought.

```
python
Copy code
prompt = """
How can I study effectively for my board exams?
Example 1: 'Set realistic daily goals.'
Example 2: 'Use flashcards for quick review.'
Now, guide me through a detailed study plan step by step.
"""
print(get_completion(prompt))
```

8. Prompt Chaining

Break down the task into smaller, linked prompts.

```
python
Copy code
# First prompt in the chain
prompt1 = "How should I create a study schedule for my board exams?"
response1 = get_completion(prompt1)
print(response1)

# Second prompt based on first response
prompt2 = "How should I revise for each subject in the schedule?"
response2 = get_completion(prompt2)
print(response2)

# Third prompt based on second response
prompt3 = "How can I test myself to check my readiness for the exam?"
```

```
response3 = get_completion(prompt3)
print(response3)
```