

Article

Not peer-reviewed version

Harnessing Generative AI in Education: From Theory to Real-World Impact

[Abhishek Verma](#)^{*} and Nallarasan V

Posted Date: 16 May 2025

doi: 10.20944/preprints202505.1177.v2

Keywords: GenAI; LLMs; PEFT; LoRA; Prompt Tuning; PPO; Reinforcement Learning; Fine-Tuning



Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This open access article is published under a Creative Commons CC BY 4.0 license, which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

Harnessing Generative AI in Education: From Theory to Real-World Impact

Abhishek Verma * and Nallarasan V

Department of Information Technology, SRMIST, Chennai, Tamil Nadu, Chennai, India

* Correspondence: av6651@srmist.edu.in

Abstract: The rise of generative artificial intelligence (GenAI) is transforming the education industry. GenAI models, particularly large language models (LLMs), have emerged as powerful tools capable of driving innovation, improving efficiency, and delivering superior services for educational purposes. This paper provides an overview of GenAI for educational purposes, from theory to practice. We have developed a chatbot for summarizing dialogues. In our research work, we have used strategies like zero-shot, one-shot, and few-shot inferencing and also fine-tuned the FLAN-T5 model to serve our purpose of summarization for educational tasks using PEFT (Parameter-Efficient Fine-Tuning) techniques like LoRA (Low-Rank Adaptation) and Prompt Tuning. We have also utilized the technique of Reinforcement Learning with PPO (Proximal Policy Optimization) and PEFT to generate less toxic summaries. The model's performance is quantitatively evaluated using the ROUGE metric and toxicity evaluation metrics. The chatbot can summarize dialogues and is of immense interest to users in the real world. In our research work, our findings demonstrate significant improvements in summarization quality and toxicity reduction, contributing to the development of safer and more effective AI systems.

Keywords: GenAI, LLMs; PEFT; LoRA; Prompt Tuning; PPO; Reinforcement Learning; fine-tuning

1. Introduction

The rise of generative artificial intelligence is powered by many factors like vast amounts of data, deep learning algorithms, transformer architecture, and high-performance computing accelerated by graphics processing units (GPUs). These technological advancements have led to the creation of powerful GenAI models, particularly large language models (LLMs) like generative pre-trained transformers (GPT). The exceptional performance of GenAI models (e.g., OpenAI's GPT-3.5 Turbo, GPT-4, and GPT-4o) and their access through user-friendly interfaces have brought text and image generation to the forefront of daily and commonplace conversations. Techniques like Prompt Engineering are involved in building application features like Deep Research, Reason, along with useful additional features in LLMs like Web Search. Now, GenAI is transforming the world, driving innovations in a wide range of industries and emerging applications.

The education industry is among the first to embrace GenAI and benefits significantly from the resources provided through applications like GenAI-powered chatbots [1]. Chatbots powered by GenAI can assist customers with their queries and troubleshoot technical issues. GenAI is helping customers get a better experience in a variety of domains related to education. Today, there are various fine-tuned LLMs that are domain-specific, like those for healthcare, which can help answer queries related to healthcare. Overall, the potential of GenAI for education is vast and will continue to grow as the technology evolves [2]. Recent research works applying a multitude of GenAI models to the education domain include an overview of the current state and future directions of generative AI in education along with applications, challenges, and research opportunities [3]. The work [4] explains how large language models like GPT-4 can enhance personalized education through dynamic content generation, real-time feedback, and adaptive learning pathways within Intelligent Tutoring. The work [5] investigates the perceived benefits and challenges of generative AI in higher education from the

perspectives of both teachers and students. It further explores concerns regarding the potential of AI to replace educators and examines its implications for digital literacy through the lenses of the SAMR and UTAUT models. The work [6] discusses the role of generative AI in education and research, emphasizing its potential to support educational goals while also addressing ethical concerns and the professional use of AI-generated content.

While existing work has explored the potential of GenAI for the education sector, there remains a gap between research outcomes and real-world applications. The existing literature primarily focuses on the theory or vision of GenAI for education, often overlooking the implications and challenges that exist in practice. To that end, we first examine the commonly used GenAI models for education by highlighting their theoretical foundations and relevance to key use cases. Then, we specifically focus on LLMs and provide an overview of the practical applications of LLMs as found in the education industry today. The developed LLM-based application utilizes innovative fine-tuning and evaluation strategies. In our research work, we have used inference strategies like zero-shot, one-shot, and few-shot techniques and fine-tuning approaches using PEFT (Parameter-Efficient Fine-Tuning) techniques like LoRA (Low-Rank Adaptation) and Prompt Tuning. We have also utilized the technique of Reinforcement Learning with PPO (Proximal Policy Optimization) and PEFT to generate less toxic summaries. The model's performance is quantitatively evaluated using the ROUGE metric and toxicity evaluation metrics. The chatbot can summarize dialogues and is of immense interest to users in the real world, leading to the development of safer and more effective AI systems.

2. Preliminaries of Generative AI Models for Education

GenAI is used to produce new but similar samples distributed according to some unknown distribution of the existing samples. The goal of GenAI modeling is to develop a model that learns the unknown distribution so that we can use it for sampling. A multitude of GenAI models have been applied to educational problems, including transformer, diffusion models (DF), variational autoencoder (VAE), generative adversarial network (GAN), and Autoregressive (AR) based models. In this section, we present the preliminaries of these GenAI models, as illustrated in Figure 1.

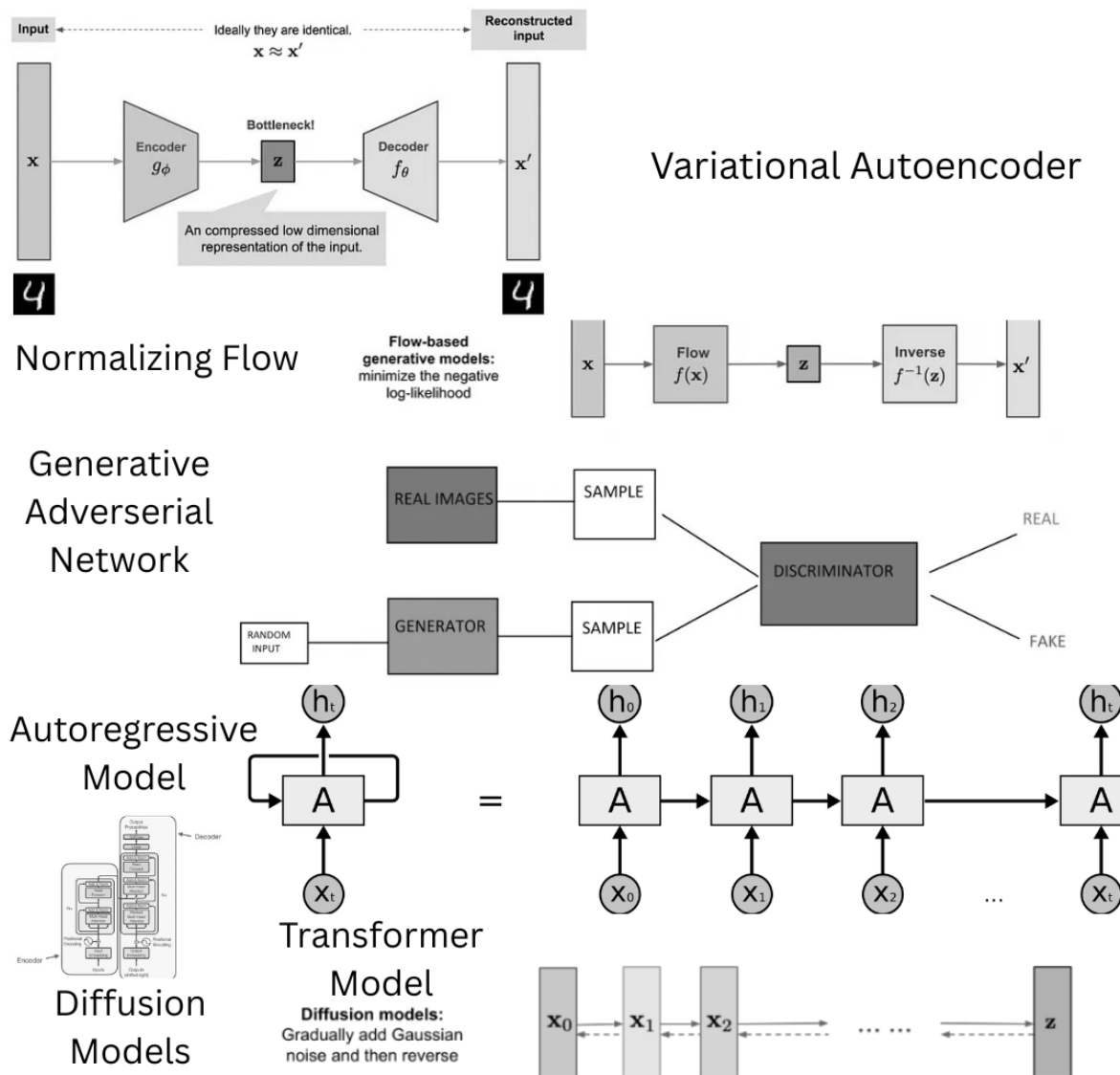


Figure 1. Overview of Generative AI Models in Education.

2.1. Variational Autoencoders

Variational Autoencoders (VAEs) are a class of generative models that learn to represent complex data distributions using a latent variable framework. They combine variational inference with deep learning by optimizing a lower bound on the data likelihood using stochastic gradient descent. The model consists of two parts: an encoder (inference network) that maps input data to a distribution over latent variables, and a decoder (generative network) that reconstructs the data from the latent space. A key innovation is the reparameterization trick, which allows backpropagation through stochastic nodes, making the training efficient and scalable with deep networks [7].

2.2. Generative Adversarial Networks

Generative Adversarial Networks (GANs) introduce a novel framework for estimating generative models via an adversarial process. This involves training two models simultaneously: a generator that captures the data distribution and a discriminator that estimates the probability that a sample came from the training data rather than the generator. The generator aims to produce data that is indistinguishable from real data, while the discriminator strives to differentiate between real and generated data. This setup corresponds to a minimax two-player game. The entire system can be trained using backpropagation without the need for Markov chains or unrolled approximate

inference networks. Experiments demonstrate the potential of this framework through qualitative and quantitative evaluations of the generated samples [8].

2.3. Normalizing Flows

Normalizing Flows enhance variational inference by applying a sequence of invertible transformations to a simple initial density, thereby constructing flexible and scalable approximate posterior distributions. This approach enables more expressive variational approximations, leading to improved inference quality in complex probabilistic models. By utilizing invertible transformations, the method allows for the modeling of more complex distributions while maintaining tractability, making it an effective tool for enhancing variational inference in various probabilistic settings [9].

2.4. Diffusion Models

Diffusion Models are a class of generative models that learn to reverse a gradual noising process, effectively generating data by iteratively denoising samples. These models have demonstrated remarkable success in generating high-quality images and have been applied to various domains, including image synthesis and inpainting. By modeling the data distribution through a diffusion process, these models can capture complex data structures and generate samples that closely resemble real-world data [10].

2.5. Autoregressive Models

Autoregressive Models are a class of generative models that generate data sequentially, with each data point conditioned on the previous ones. This approach allows for the modeling of complex dependencies in data, making these models particularly effective for tasks such as language modeling, speech synthesis, and image generation. By capturing the conditional distributions of data points, autoregressive models can generate high-quality samples that reflect the underlying data distribution [11].

The different types of GenAI models have varying levels of performance in terms of the quality of generation outputs, the diversity of mode coverage, and the speed of sampling. Combining the advantages of the GenAI models, when possible, can create more powerful GenAI models for education. While GenAI models have shown great potential for enabling various emerging educational use cases in simulated or lab environments, transformer-based LLMs are among the most popular GenAI models that are already finding practical applications in the current education industry. These real-world applications of LLMs in the education domain are detailed in the next section.

3. LLMs for Education

Large Language Models (LLMs) have become pivotal in educational applications due to their ability to process and generate human-like text. This section explores their practical implementations, focusing both on theoretical advancements and real-world deployments.

3.1. Student Support and Personalized Learning

Domain-specific LLMs are transforming academic support and personalized education. LLM-based tutoring platforms assist students in real-time with subject-specific doubts, curate personalized learning plans based on performance history, and generate explanations aligned with individual understanding levels. These systems can also engage in natural language check-ins to ensure student progress and offer targeted supplemental materials.

Platforms such as Khanmigo by Khan Academy illustrate these capabilities. They employ fine-tuned LLMs to offer interactive AI tutors that explain mathematical problems step-by-step or simulate historical conversations for immersive learning. These tutors are grounded in educational content and best pedagogical practices to provide contextual and adaptive support.

Moreover, LLM-powered virtual academic advisors enhance student engagement by interpreting behavioral signals, summarizing professor feedback, and initiating proactive interventions to maintain academic progress.

3.2. Faculty and Administrative Assistance

LLMs streamline repetitive academic and administrative tasks, increasing educator productivity. These assistants can draft lesson plans, summarize research articles, generate quizzes from textbook material, and communicate professionally with students and parents.

For instance, Microsoft's Copilot for Education, integrated into platforms like Microsoft Teams and Word, already helps educators create assignments, summarize lectures, and design assessments aligned with curriculum standards.

In parallel, LLM-powered bots handle routine administrative inquiries—such as course registration, deadlines, and financial aid—thus reducing staff workload and enhancing the student experience.

3.3. Curriculum Planning and Educational Analytics

Educational leaders are leveraging LLMs to interact with academic data through natural language. These systems analyze student performance, detect learning gaps, and provide actionable insights for curriculum enhancement.

Tools like Ivy.ai and Gradescope with AI assistance support real-time feedback, pattern recognition in assignment submissions, and identification of learning bottlenecks. They can recommend course redesign strategies and prioritize interventions using longitudinal performance data.

Generative AI can also simulate new learning environments using synthetic data, supporting curriculum development and strategic planning.

3.4. Educational Standards Chatbots and Knowledge Access

Comprehending evolving academic standards (e.g., Common Core, NGSS, Bloom's Taxonomy) across disciplines is complex. LLM-based systems, especially those powered by Retrieval-Augmented Generation (RAG), simplify access to these documents, aiding educators and curriculum developers.

RAG-powered curriculum alignment bots enable teachers to instantly understand how lesson objectives relate to standards or compare regional benchmarks. These tools support transparency, alignment, and compliance with evolving frameworks.

Additionally, LLMs support inclusive education by translating content across languages, simplifying academic jargon, and adapting materials for diverse learner needs.

3.5. Research Implementation: Dialogue Summarization Chatbot

Our own research demonstrates a practical implementation of LLMs in education through the development of a dialogue summarization chatbot. This chatbot leverages zero-shot, one-shot, and few-shot inferencing with fine-tuned FLAN-T5 using PEFT techniques such as LoRA and Prompt Tuning. We also employed Reinforcement Learning with Proximal Policy Optimization (PPO) to reduce toxic content in the generated summaries. The system is quantitatively evaluated using the ROUGE metric and toxicity evaluation tools, achieving strong results in both summarization quality and safety, thereby supporting real-world classroom integration.

4. Design Aspects

The development and deployment of generative AI (GenAI) applications, particularly large language models (LLMs), require careful consideration of design aspects to ensure they meet the specific requirements of the education sector. While cloud-based GenAI services offer a rapid entry point, their general-purpose nature often lacks the depth of training on education-specific datasets, limiting their effectiveness for targeted educational applications. This section discusses key design aspects for building customized AI applications tailored to educational needs, as illustrated in Figure 2.

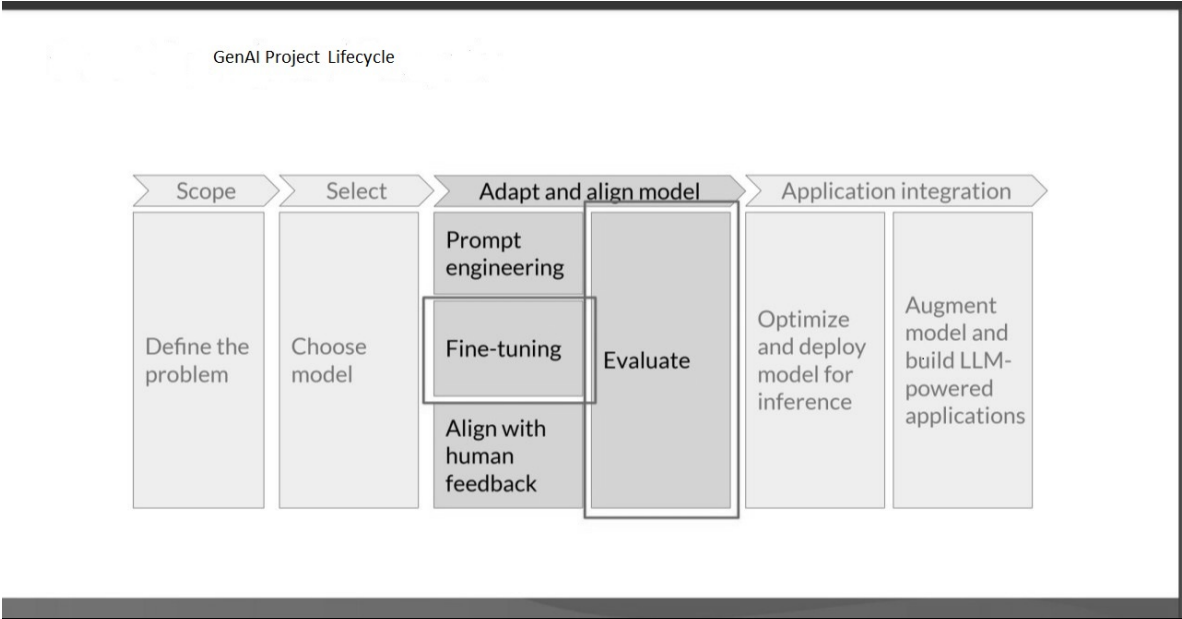


Figure 2. An overview of the life cycle of building GenAI applications for educational contexts, such as AI tutors and curriculum chatbots.

4.1. High-Performance Computing for AI

The transformer architecture, foundational to modern large language models (LLMs), demands substantial computational resources due to its intricate design. Unlike other AI systems, which have seen relatively modest growth in compute needs, transformer-based models experience exponential increases in resource requirements, driven by their complex layers and vast parameter sets. In educational environments, where processing large volumes of data—such as student engagement metrics, instructional content, or real-time feedback—is essential, high-performance hardware like multi-GPU clusters is critical for training and deploying these models effectively.

Advanced GPUs, such as NVIDIA’s latest architectures, provide tailored optimizations that enhance training efficiency, minimize memory usage, and enable scalable AI infrastructure. For instance, these GPUs support low-precision formats like FP8, which accelerate computations but risk degrading model quality if not carefully managed. To address this, modern GPU architectures incorporate mixed-precision engines that dynamically balance FP8 and FP16 operations, automatically adjusting to preserve accuracy. This capability is particularly valuable for educational tools, such as AI-driven essay grading systems or virtual tutoring platforms, where rapid inference ensures seamless user interactions.

In academic settings, high-performance computing systems enable efficient analysis of extensive datasets, such as course enrollment patterns, learning management system logs, or multilingual educational resources. Compared to traditional CPU-based setups, these platforms consume less energy, promoting sustainable operations. For example, AI-powered language translation tools for diverse classrooms or predictive analytics for student retention require low-latency processing to deliver timely insights, enhancing both teaching quality and administrative efficiency. As educational institutions increasingly adopt AI-driven solutions, high-performance computing will drive innovation, ensuring scalability without computational limitations.

4.2. Information Retrieval and Customization

While foundation LLMs are pre-trained on vast datasets using accelerated computing, their effectiveness in education depends on incorporating domain-specific knowledge and institutional data. Techniques like Retrieval-Augmented Generation (RAG), prompt engineering, and parameter-efficient fine-tuning (PEFT) enable customization to meet educational needs.

Retrieval-Augmented Generation (RAG): RAG enhances LLMs by integrating external data sources, improving the accuracy and relevance of responses. Figure 3 illustrates a typical RAG architecture model for educational applications, such as curriculum standards chatbots:

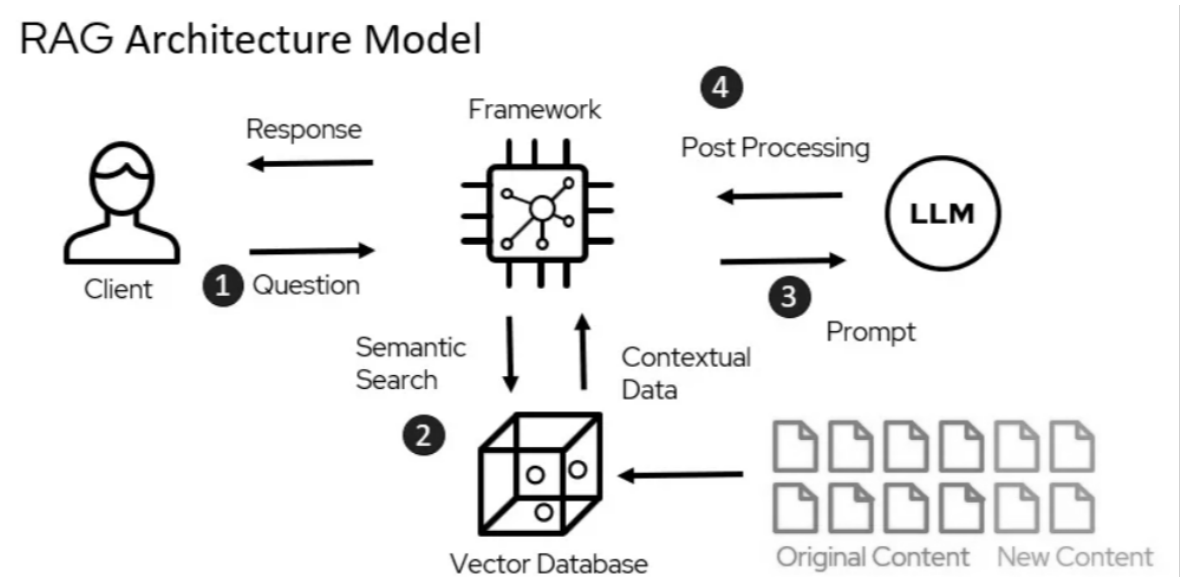


Figure 3. RAG architecture model illustrating the flow of information retrieval and response generation for educational applications.

- **Step 1 (Question Submission):** The process begins with a client, such as a teacher or student, submitting a question (e.g., querying a curriculum standard or seeking topic clarification).
- **Step 2 (Semantic Search):** The question triggers a semantic search in a vector database, which stores embeddings of educational resources, including both original content (e.g., textbooks, academic standards) and new content (e.g., updated lesson plans or student data).
- **Step 3 (Prompt Construction):** Relevant contextual data is retrieved from the vector database and used to construct a prompt, which is then fed into a Large Language Model (LLM).
- **Step 4 (Response Generation and Post-Processing):** The LLM generates a response based on the prompt, which undergoes post-processing within a framework to ensure coherence and relevance before being delivered back to the client.

For example, a curriculum standards chatbot can use RAG to integrate educational standards (e.g., Common Core) into an LLM, enabling precise answers to domain-specific queries. This approach is particularly effective for applications like lesson planning, student support systems, or compliance with educational frameworks.

Customization Techniques: Beyond RAG, techniques like PEFT (e.g., LoRA and Prompt Tuning) and reinforcement learning with human feedback (RLHF) allow further customization. PEFT selectively updates a small subset of model parameters, making fine-tuning resource-efficient while adapting the LLM to educational datasets, such as student feedback, course materials, or assessment records. RLHF, combined with techniques like Proximal Policy Optimization (PPO), can reduce toxicity in generated outputs, as demonstrated in our dialogue summarization chatbot for classroom discussions, ensuring safer and more appropriate interactions in educational settings.

RAG and fine-tuning are complementary. RAG offers a quick way to enhance accuracy by grounding responses in external data, while fine-tuning provides deeper customization for applications requiring high precision, such as personalized learning systems or automated grading tools. The choice of approach depends on the application’s requirements, resource availability, and computational constraints. For instance, a chatbot for summarizing classroom dialogues can initially use RAG to incorporate lecture transcripts and later apply PEFT for improved performance, as shown in our research implementation.

5. Case Study: Dialog Summarization Chatbot

In our research, we have used inference techniques such as zero-shot, one-shot, and few-shot to evaluate the model’s performance on dialogue summarization tasks. These techniques are demonstrated through various implementations, as shown in the figures below. Additionally, we have applied fine-tuning methods including LoRA (Low-Rank Adaptation), PEFT (Parameter-Efficient Fine-Tuning), Prompt Tuning, and Reinforcement Learning with PPO (Proximal Policy Optimization) to enhance the model’s capabilities. The following figures illustrate the implementation details, code, and outputs of these techniques.

Summarize Dialogue without Prompt Engineering



Figure 4. Dialogue summarization without prompt engineering (Part 1).

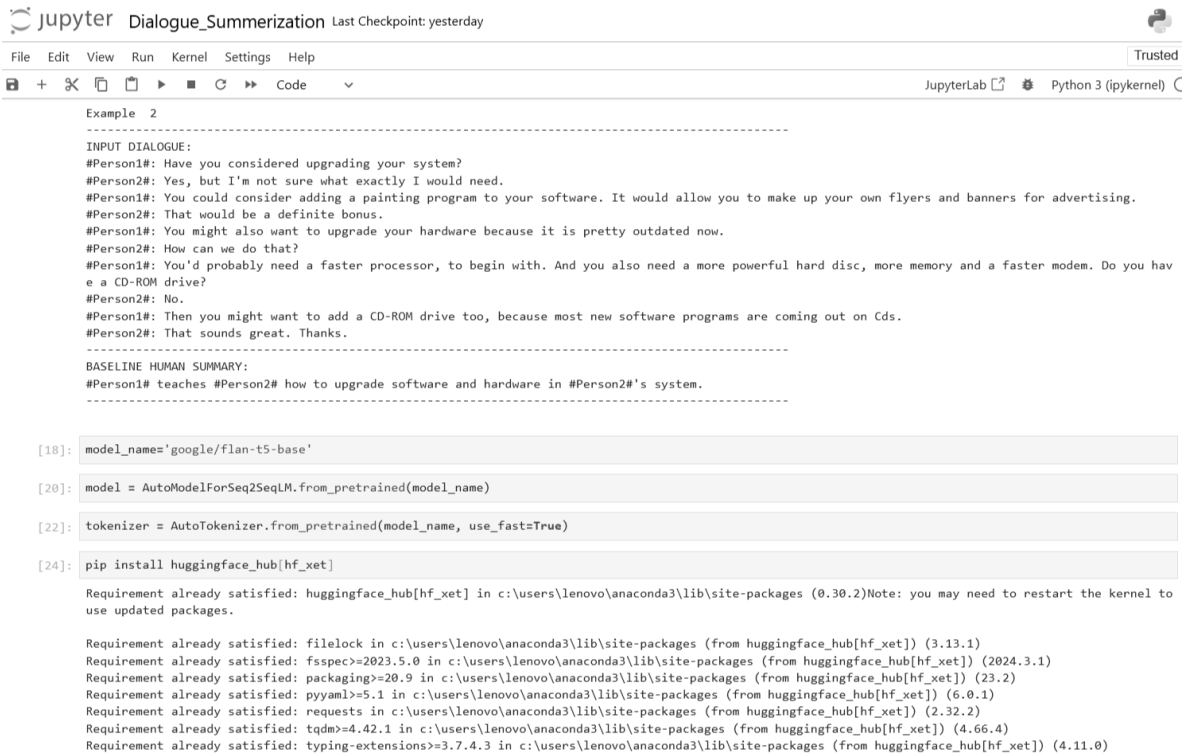


Figure 5. Dialogue summarization without prompt engineering (Part 2).

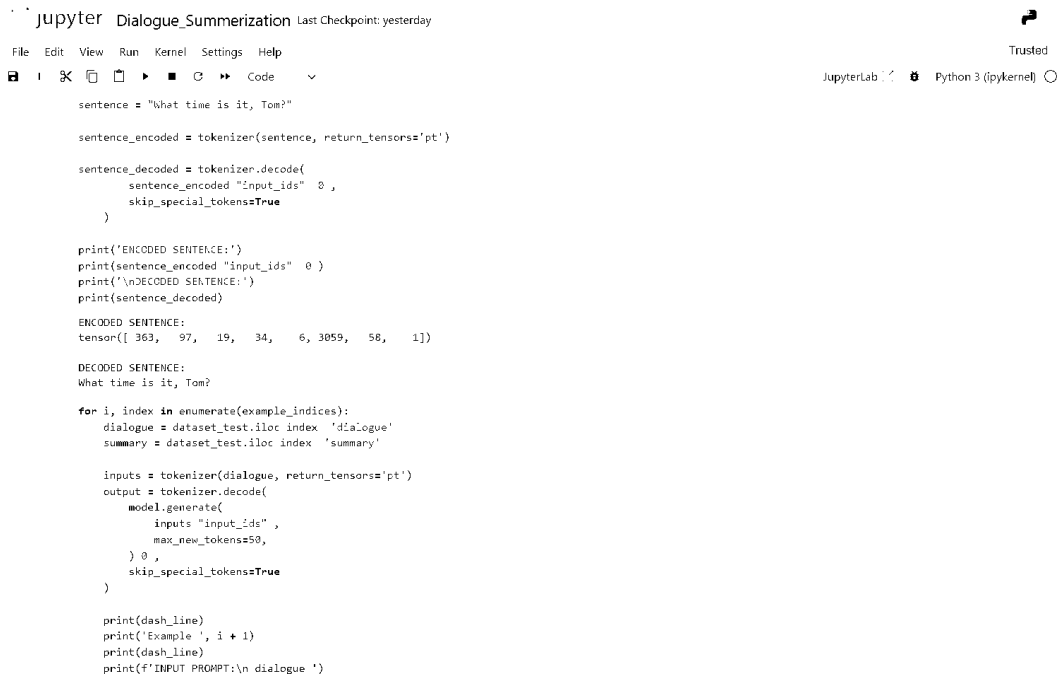


Figure 6. Encode and decode string implementation.



Figure 7. Model generation without prompt engineering (Part 1).

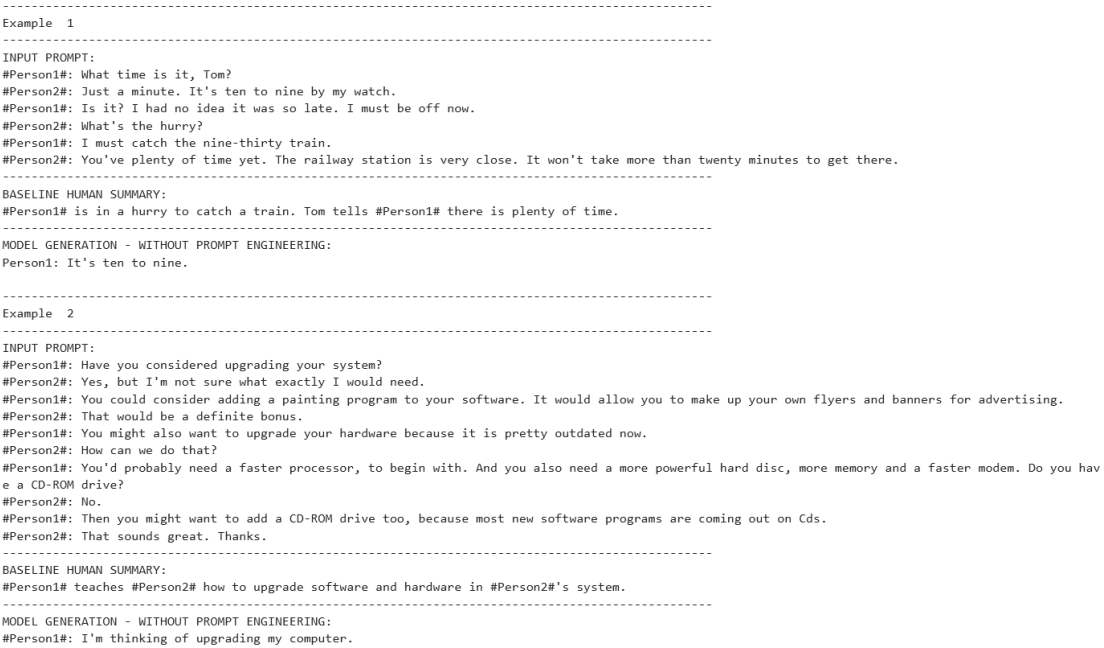


Figure 8. Model generation without prompt engineering (Part 2).

Summarize Dialogue with an Instruction Prompt

Zero Shot Inference with an Instruction Prompt

```
[38]: for i, index in enumerate(example_indices):
    dialogue = dataset_test.iloc[index]['dialogue']
    summary = dataset_test.iloc[index]['summary']

    prompt = f"""
Summarize the following conversation.

{dialogue}

Summary:
"""

    # Input constructed prompt instead of the dialogue.
    inputs = tokenizer(prompt, return_tensors='pt')
    output = tokenizer.decode(
        model.generate(
            inputs["input_ids"],
            max_new_tokens=50,
        )[0],
        skip_special_tokens=True
    )

    print(dash_line)
    print('Example ', i + 1)
    print(dash_line)
    print(f'INPUT PROMPT:\n{prompt}')
    print(dash_line)
    print(f'BASELINE HUMAN SUMMARY:\n{summary}')
    print(dash_line)
    print(f'MODEL GENERATION - ZERO SHOT:\n{output}\n')
```

Figure 9. Zero-shot inference instruction prompt.

```
[63]: for i, index in enumerate(example_indices):
    dialogue = dataset_test.iloc[index]['dialogue']
    summary = dataset_test.iloc[index]['summary']

    prompt = f"""
Summarize the following conversation.

{dialogue}

"""

    # Input constructed prompt instead of the dialogue.
    inputs = tokenizer(prompt, return_tensors='pt')
    output = tokenizer.decode(
        model.generate(
            inputs["input_ids"],
            max_new_tokens=50,
        )[0],
        skip_special_tokens=True
    )

    print(dash_line)
    print('Example ', i + 1)
    print(dash_line)
    print(f'INPUT PROMPT:\n{prompt}')
    print(dash_line)
    print(f'BASELINE HUMAN SUMMARY:\n{summary}')
    print(dash_line)
    print(f'MODEL GENERATION - ZERO SHOT:\n{output}\n')
```

Example 1

INPUT PROMPT:

Summarize the following conversation.

Figure 10. Zero-shot inference code (Part 1).

jupyter Dialogue_Summerization Last Checkpoint: yesterday

File Edit View Run Kernel Settings Help Trusted

JupyterLab Python 3 (ipykernel)

```
[65]: for i, index in enumerate(example_indices):
      dialogue = dataset_test.iloc[index]['dialogue']
      summary = dataset_test.iloc[index]['summary']

      prompt = f"""
      Put briefly the following conversation.

      {dialogue}

      Summary:
      """

      # Input constructed prompt instead of the dialogue.
      inputs = tokenizer(prompt, return_tensors='pt')
      output = tokenizer.decode(
          model.generate(
              inputs["input_ids"],
              max_new_tokens=50,
          )[0],
          skip_special_tokens=True
      )

      print(dash_line)
      print('Example ', i + 1)
      print(dash_line)
      print(f'INPUT PROMPT:\n{prompt}')
      print(dash_line)
      print(f'BASELINE HUMAN SUMMARY:\n{summary}')
      print(dash_line)
      print(f'MODEL GENERATION - ZERO SHOT:\n{output}\n')

      -----
      Example 1
      -----
      INPUT PROMPT:
```

Figure 11. Zero-shot inference code (Part 2).

jupyter Dialogue_Summerization Last Checkpoint: yesterday

File Edit View Run Kernel Settings Help Trusted

JupyterLab Python 3 (ipykernel)

```
-----
Example 1
-----
INPUT PROMPT:

Put briefly the following conversation.

#Person1#: What time is it, Tom?
#Person2#: Just a minute. It's ten to nine by my watch.
#Person1#: Is it? I had no idea it was so late. I must be off now.
#Person2#: What's the hurry?
#Person1#: I must catch the nine-thirty train.
#Person2#: You've plenty of time yet. The railway station is very close. It won't take more than twenty minutes to get there.

Summary:

-----
BASELINE HUMAN SUMMARY:
#Person1# is in a hurry to catch a train. Tom tells #Person1# there is plenty of time.
-----
MODEL GENERATION - ZERO SHOT:
The train is about to leave.

-----
Example 2
-----
INPUT PROMPT:

Put briefly the following conversation.

#Person1#: Have you considered upgrading your system?
#Person2#: Yes, but I'm not sure what exactly I would need.
#Person1#: You could consider adding a painting program to your software. It would allow you to make up your own flyers and banners for advertising.
#Person2#: That would be a definite bonus.
#Person1#: You might also want to upgrade your hardware because it is pretty outdated now.
#Person2#: How can we do that?
#Person1#: You'd probably need a faster processor, to begin with. And you also need a more powerful hard disc, more memory and a faster modem. Do you hav
e a CD-ROM drive?
#Person2#: No.
```

Figure 12. Zero-shot inference code (Part 3).

jupyter Dialogue_Summerization Last Checkpoint: yesterday

File Edit View Run Kernel Settings Help Trusted

JupyterLab Python 3 (ipykernel)

```
-----
Example 2
-----
INPUT PROMPT:

Put briefly the following conversation.

#Person1#: Have you considered upgrading your system?
#Person2#: Yes, but I'm not sure what exactly I would need.
#Person1#: You could consider adding a painting program to your software. It would allow you to make up your own flyers and banners for advertising.
#Person2#: That would be a definite bonus.
#Person1#: You might also want to upgrade your hardware because it is pretty outdated now.
#Person2#: How can we do that?
#Person1#: You'd probably need a faster processor, to begin with. And you also need a more powerful hard disc, more memory and a faster modem. Do you have a CD-ROM drive?
#Person2#: No.
#Person1#: Then you might want to add a CD-ROM drive too, because most new software programs are coming out on Cds.
#Person2#: That sounds great. Thanks.

Summary:

-----
BASELINE HUMAN SUMMARY:
#Person1# teaches #Person2# how to upgrade software and hardware in #Person2#'s system.
-----
MODEL GENERATION - ZERO SHOT:
#Person1#: I'm thinking of upgrading my computer.

[67]: for i, index in enumerate(example_indices):
        dialogue = dataset_test.iloc[index]['dialogue']
        summary = dataset_test.iloc[index]['summary']

        prompt = f"""
Put briefly the following conversation.

{dialogue}

"""
```

Figure 13. Zero-shot inference code (Part 4).

jupyter Dialogue_Summerization Last Checkpoint: yesterday

File Edit View Run Kernel Settings Help Trusted

JupyterLab Python 3 (ipykernel)

```
[67]: for i, index in enumerate(example_indices):
        dialogue = dataset_test.iloc[index]['dialogue']
        summary = dataset_test.iloc[index]['summary']

        prompt = f"""
Put briefly the following conversation.

{dialogue}

"""

        # Input constructed prompt instead of the dialogue.
        inputs = tokenizer(prompt, return_tensors='pt')
        output = tokenizer.decode(
            model.generate(
                inputs["input_ids"],
                max_new_tokens=50,
            )[0],
            skip_special_tokens=True
        )

        print(dash_line)
        print('Example ', i + 1)
        print(dash_line)
        print(f'INPUT PROMPT:\n{prompt}')
        print(dash_line)
        print(f'BASELINE HUMAN SUMMARY:\n{summary}')
        print(dash_line)
        print(f'MODEL GENERATION - ZERO SHOT:\n{output}\n')

-----
Example 1
-----
INPUT PROMPT:

Put briefly the following conversation
```

Figure 14. Zero-shot inference code (Part 5).

```
Example 1
-----
INPUT PROMPT:

Put briefly the following conversation.

#Person1#: What time is it, Tom?
#Person2#: Just a minute. It's ten to nine by my watch.
#Person1#: Is it? I had no idea it was so late. I must be off now.
#Person2#: What's the hurry?
#Person1#: I must catch the nine-thirty train.
#Person2#: You've plenty of time yet. The railway station is very close. It won't take more than twenty minutes to get there.

-----
BASELINE HUMAN SUMMARY:
#Person1# is in a hurry to catch a train. Tom tells #Person1# there is plenty of time.
-----
MODEL GENERATION - ZERO SHOT:
#Person1#: I'm sorry, Tom. I'm sorry to hear that.

-----
Example 2
-----
INPUT PROMPT:

Put briefly the following conversation.

#Person1#: Have you considered upgrading your system?
#Person2#: Yes, but I'm not sure what exactly I would need.
#Person1#: You could consider adding a painting program to your software. It would allow you to make up your own flyers and banners for advertising.
#Person2#: That would be a definite bonus.
#Person1#: You might also want to upgrade your hardware because it is pretty outdated now.
#Person2#: How can we do that?
#Person1#: You'd probably need a faster processor, to begin with. And you also need a more powerful hard disc, more memory and a faster modem. Do you hav
e a CD-ROM drive?
#Person2#: No.
#Person1#: Then you might want to add a CD-ROM drive too, because most new software programs are coming out on Cds.
```

Figure 15. Zero-shot inference code (Part 6).

Jupyter

Dialogue_Summerization

Last Checkpoint: yesterday

File

Edit

View

Run

Kernel

Settings

Help

Trusted

Code

JupyterLab Python 3 (ipykernel)

```
-----
Example 2
-----
INPUT PROMPT:

Put briefly the following conversation.

#Person1#: Have you considered upgrading your system?
#Person2#: Yes, but I'm not sure what exactly I would need.
#Person1#: You could consider adding a painting program to your software. It would allow you to make up your own flyers and banners for advertising.
#Person2#: That would be a definite bonus.
#Person1#: You might also want to upgrade your hardware because it is pretty outdated now.
#Person2#: How can we do that?
#Person1#: You'd probably need a faster processor, to begin with. And you also need a more powerful hard disc, more memory and a faster modem. Do you hav
e a CD-ROM drive?
#Person2#: No.
#Person1#: Then you might want to add a CD-ROM drive too, because most new software programs are coming out on Cds.
#Person2#: That sounds great. Thanks.

-----
BASELINE HUMAN SUMMARY:
#Person1# teaches #Person2# how to upgrade software and hardware in #Person2#'s system.
-----
MODEL GENERATION - ZERO SHOT:
#Person1#: Thanks.

[40]: for i, index in enumerate(example_indices):
        dialogue = dataset_test.iloc[index]['dialogue']
        summary = dataset_test.iloc[index]['summary']

        prompt = f"""
Dialogue:

{dialogue}

What was going on?
```

Figure 16. Zero-shot inference code (Part 7).

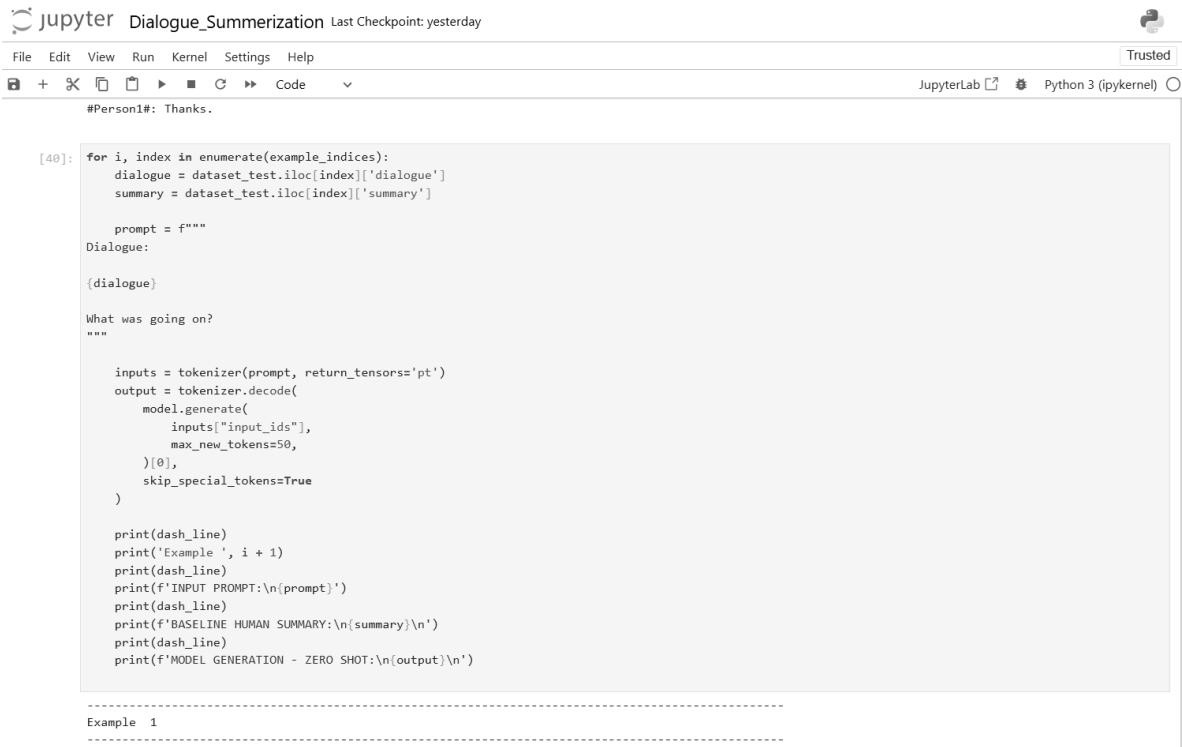


Figure 17. Zero-shot inference code (Part 8).

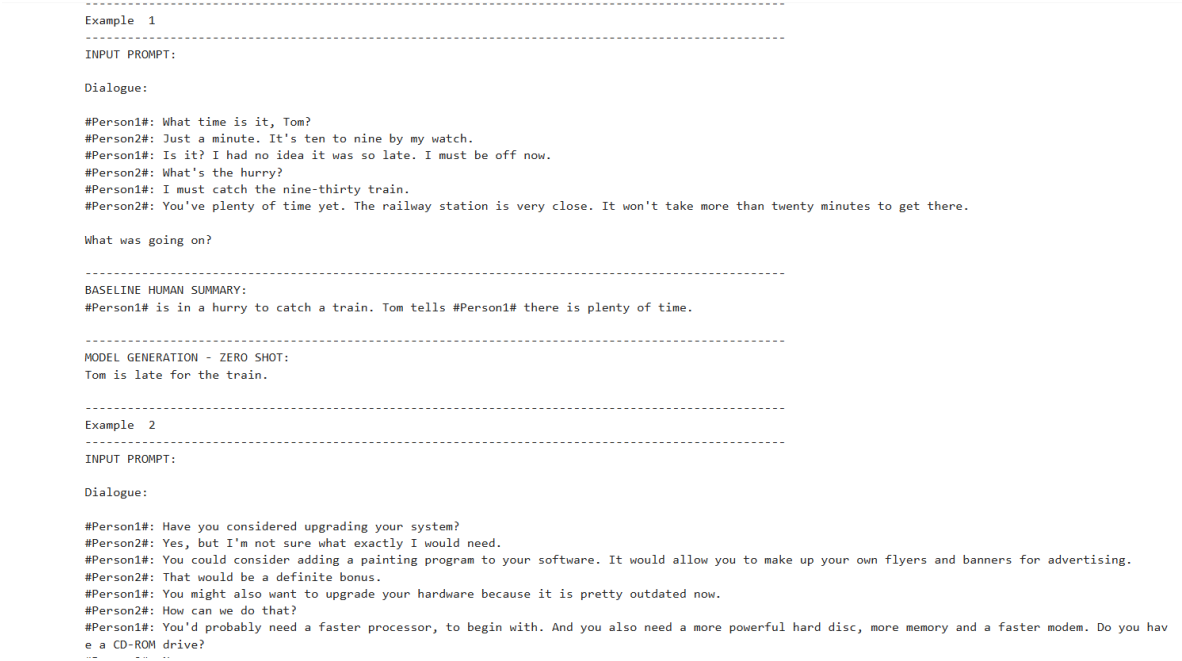


Figure 18. Zero-shot inference code (Part 9).

```
-----
BASELINE HUMAN SUMMARY:
#Person1# is in a hurry to catch a train. Tom tells #Person1# there is plenty of time.
-----

MODEL GENERATION - ZERO SHOT:
Tom is late for the train.
-----

Example 2
-----
INPUT PROMPT:

Dialogue:

#Person1#: Have you considered upgrading your system?
#Person2#: Yes, but I'm not sure what exactly I would need.
#Person1#: You could consider adding a painting program to your software. It would allow you to make up your own flyers and banners for advertising.
#Person2#: That would be a definite bonus.
#Person1#: You might also want to upgrade your hardware because it is pretty outdated now.
#Person2#: How can we do that?
#Person1#: You'd probably need a faster processor, to begin with. And you also need a more powerful hard disc, more memory and a faster modem. Do you have a CD-ROM drive?
#Person2#: No.
#Person1#: Then you might want to add a CD-ROM drive too, because most new software programs are coming out on Cds.
#Person2#: That sounds great. Thanks.

What was going on?
-----

BASELINE HUMAN SUMMARY:
#Person1# teaches #Person2# how to upgrade software and hardware in #Person2#'s system.
-----

MODEL GENERATION - ZERO SHOT:
#Person1#: You could add a painting program to your software. #Person2#: That would be a bonus. #Person1#: You might also want to upgrade your hardware.
#Person1#
```

Figure 19. Zero-shot inference code (Part 10).

Zero Shot Inference with the Prompt Template from FLAN-T5

```
[72]: for i, index in enumerate(example_indices):
    dialogue = dataset_test.iloc[index]['dialogue']
    summary = dataset_test.iloc[index]['summary']

    prompt = f"""
Summarize the following conversation.

{dialogue}

"""

    # Input constructed prompt instead of the dialogue.
    inputs = tokenizer(prompt, return_tensors='pt')
    output = tokenizer.decode(
        model.generate(
            inputs["input_ids"],
            max_new_tokens=50,
        )[0],
        skip_special_tokens=True
    )

    print(dash_line)
    print('Example ', i + 1)
    print(dash_line)
    print(f'INPUT PROMPT:\n{prompt}')
    print(dash_line)
    print(f'BASELINE HUMAN SUMMARY:\n{summary}')
    print(dash_line)
    print(f'MODEL GENERATION - ZERO SHOT:\n{output}\n')

-----
Example 1
-----
```

Figure 20. Zero-shot inference code (Part 11).

jupyter Dialogue_Summerization Last Checkpoint: yesterday

File Edit View Run Kernel Settings Help

Trusted

JupyterLab Python 3 (ipykernel)

```
-----
Example 1
-----
INPUT PROMPT:

Summarize the following conversation.

#Person1#: What time is it, Tom?
#Person2#: Just a minute. It's ten to nine by my watch.
#Person1#: Is it? I had no idea it was so late. I must be off now.
#Person2#: What's the hurry?
#Person1#: I must catch the nine-thirty train.
#Person2#: You've plenty of time yet. The railway station is very close. It won't take more than twenty minutes to get there.

-----
BASELINE HUMAN SUMMARY:
#Person1# is in a hurry to catch a train. Tom tells #Person1# there is plenty of time.
-----
MODEL GENERATION - ZERO SHOT:
#Person1#: It's ten to nine.
-----

Example 2
-----
INPUT PROMPT:

Summarize the following conversation.

#Person1#: Have you considered upgrading your system?
#Person2#: Yes, but I'm not sure what exactly I would need.
#Person1#: You could consider adding a painting program to your software. It would allow you to make up your own flyers and banners for advertising.
#Person2#: That would be a definite bonus.
#Person1#: You might also want to upgrade your hardware because it is pretty outdated now.
#Person2#: How can we do that?
#Person1#: You'd probably need a faster processor, to begin with. And you also need a more powerful hard disc, more memory and a faster modem. Do you have a CD-ROM drive?
#Person2#: No.
#Person1#: Then you might want to add a CD-ROM drive too, because most new software programs are coming out on Cds.
```

Figure 21. Zero-shot inference input and output (Part 1).

jupyter Dialogue_Summerization Last Checkpoint: yesterday

File Edit View Run Kernel Settings Help

Trusted

JupyterLab Python 3 (ipykernel)

```
Example 2
-----
INPUT PROMPT:

Summarize the following conversation.

#Person1#: Have you considered upgrading your system?
#Person2#: Yes, but I'm not sure what exactly I would need.
#Person1#: You could consider adding a painting program to your software. It would allow you to make up your own flyers and banners for advertising.
#Person2#: That would be a definite bonus.
#Person1#: You might also want to upgrade your hardware because it is pretty outdated now.
#Person2#: How can we do that?
#Person1#: You'd probably need a faster processor, to begin with. And you also need a more powerful hard disc, more memory and a faster modem. Do you have a CD-ROM drive?
#Person2#: No.
#Person1#: Then you might want to add a CD-ROM drive too, because most new software programs are coming out on Cds.
#Person2#: That sounds great. Thanks.

-----
BASELINE HUMAN SUMMARY:
#Person1# teaches #Person2# how to upgrade software and hardware in #Person2#'s system.
-----
MODEL GENERATION - ZERO SHOT:
#Person1#: I'm thinking of upgrading my computer.
```

```
[65]: for i, index in enumerate(example_indices):
      dialogue = dataset_test.iloc[index]['dialogue']
      summary = dataset_test.iloc[index]['summary']

      prompt = f"""
      Put briefly the following conversation.

      {dialogue}

      Summary:
      """
```

Figure 22. Zero-shot inference input and output (Part 2).

Jupyter Dialogue_Summerization Last Checkpoint: yesterday

File Edit View Run Kernel Settings Help

Trusted

JupyterLab Python 3 (ipykernel)

```
-----
Example 1
-----
INPUT PROMPT:

Summarize the following conversation.

#Person1#: What time is it, Tom?
#Person2#: Just a minute. It's ten to nine by my watch.
#Person1#: Is it? I had no idea it was so late. I must be off now.
#Person2#: What's the hurry?
#Person1#: I must catch the nine-thirty train.
#Person2#: You've plenty of time yet. The railway station is very close. It won't take more than twenty minutes to get there.

-----
BASELINE HUMAN SUMMARY:
#Person1# is in a hurry to catch a train. Tom tells #Person1# there is plenty of time.
-----
MODEL GENERATION - ZERO SHOT:
#Person1#: It's ten to nine.
-----

Example 2
-----
INPUT PROMPT:

Summarize the following conversation.

#Person1#: Have you considered upgrading your system?
#Person2#: Yes, but I'm not sure what exactly I would need.
#Person1#: You could consider adding a painting program to your software. It would allow you to make up your own flyers and banners for advertising.
#Person2#: That would be a definite bonus.
#Person1#: You might also want to upgrade your hardware because it is pretty outdated now.
#Person2#: How can we do that?
#Person1#: You'd probably need a faster processor, to begin with. And you also need a more powerful hard disc, more memory and a faster modem. Do you have a CD-ROM drive?
#Person2#: No.
#Person1#: Then you might want to add a CD-ROM drive too, because most new software programs are coming out on Cds.
```

Figure 23. Zero-shot inference input and output (Part 3).

Jupyter Dialogue_Summerization Last Checkpoint: yesterday

File Edit View Run Kernel Settings Help

Trusted

JupyterLab Python 3 (ipykernel)

```
-----
Example 2
-----
INPUT PROMPT:

Summarize the following conversation.

#Person1#: Have you considered upgrading your system?
#Person2#: Yes, but I'm not sure what exactly I would need.
#Person1#: You could consider adding a painting program to your software. It would allow you to make up your own flyers and banners for advertising.
#Person2#: That would be a definite bonus.
#Person1#: You might also want to upgrade your hardware because it is pretty outdated now.
#Person2#: How can we do that?
#Person1#: You'd probably need a faster processor, to begin with. And you also need a more powerful hard disc, more memory and a faster modem. Do you have a CD-ROM drive?
#Person2#: No.
#Person1#: Then you might want to add a CD-ROM drive too, because most new software programs are coming out on Cds.
#Person2#: That sounds great. Thanks.

-----
BASELINE HUMAN SUMMARY:
#Person1# teaches #Person2# how to upgrade software and hardware in #Person2#'s system.
-----
MODEL GENERATION - ZERO SHOT:
#Person1#: I'm thinking of upgrading my computer.

[46]: def make_prompt(example_indices_full, example_index_to_summarize):
    prompt = ''
    for index in example_indices_full:
        dialogue = dataset_test.iloc[index]['dialogue']
        summary = dataset_test.iloc[index]['summary']

        # The stop sequence '{summary}\n\n' is important for FLAN-T5. Other models may have their own preferred stop sequence.
        prompt += f"""
Dialogue:
{dialogue}
```

Figure 24. Zero-shot inference input and output (Part 4).

```
Summary:
"""

inputs = tokenizer(prompt, return_tensors='pt')
output = tokenizer.decode(
    original_model.generate(
        inputs["input_ids"],
        max_new_tokens=200,
    )[0],
    skip_special_tokens=True
)

dash_line = '-' * 100
print(dash_line)
print(f'INPUT PROMPT: \n{prompt}')
print(dash_line)
print(f'BASELINE HUMAN SUMMARY: \n{summary}\n')
print(dash_line)
print(f'MODEL GENERATION - ZERO SHOT: \n{output}')

-----
INPUT PROMPT:

Summarize the following conversation.

#Person1#: Have you considered upgrading your system?
#Person2#: Yes, but I'm not sure what exactly I would need.
#Person1#: You could consider adding a painting program to your software. It would allow you to make up your own flyers and banners for advertising.
#Person2#: That would be a definite bonus.
#Person1#: You might also want to upgrade your hardware because it is pretty outdated now.
#Person2#: How can we do that?
#Person1#: You'd probably need a faster processor, to begin with. And you also need a more powerful hard disc, more memory and a faster modem. Do you have a CD-ROM drive?
#Person2#: No.
#Person1#: Then you might want to add a CD-ROM drive too, because most new software programs are coming out on Cds.
#Person2#: That sounds great. Thanks.

Summary:

-----
BASELINE HUMAN SUMMARY:
#Person1# teaches #Person2# how to upgrade software and hardware in #Person2#'s system.

-----
MODEL GENERATION - ZERO SHOT:
#Person1#: I'm thinking of upgrading my computer.
```

Figure 25. Zero-shot inference input and output (Part 5).

Jupyter Dialogue_Summerization Last Checkpoint: yesterday

File Edit View Run Kernel Settings Help

Trusted

JupyterLab Python 3 (ipykernel)

```
Test the Model with Zero Shot Inferencing

[202]: index = 200

dialogue = dataset_test.iloc[index]['dialogue']
summary = dataset_test.iloc[index]['summary']

prompt = f"""
Summarize the following conversation.

{dialogue}

Summary:
"""

inputs = tokenizer(prompt, return_tensors='pt')
output = tokenizer.decode(
    original_model.generate(
        inputs["input_ids"],
        max_new_tokens=200,
    )[0],
    skip_special_tokens=True
)

dash_line = '-' * 100
print(dash_line)
print(f'INPUT PROMPT: \n{prompt}')
print(dash_line)
print(f'BASELINE HUMAN SUMMARY: \n{summary}\n')
print(dash_line)
print(f'MODEL GENERATION - ZERO SHOT: \n{output}')

-----
INPUT PROMPT:
```

Figure 26. Zero-shot inference testing (Part 1).

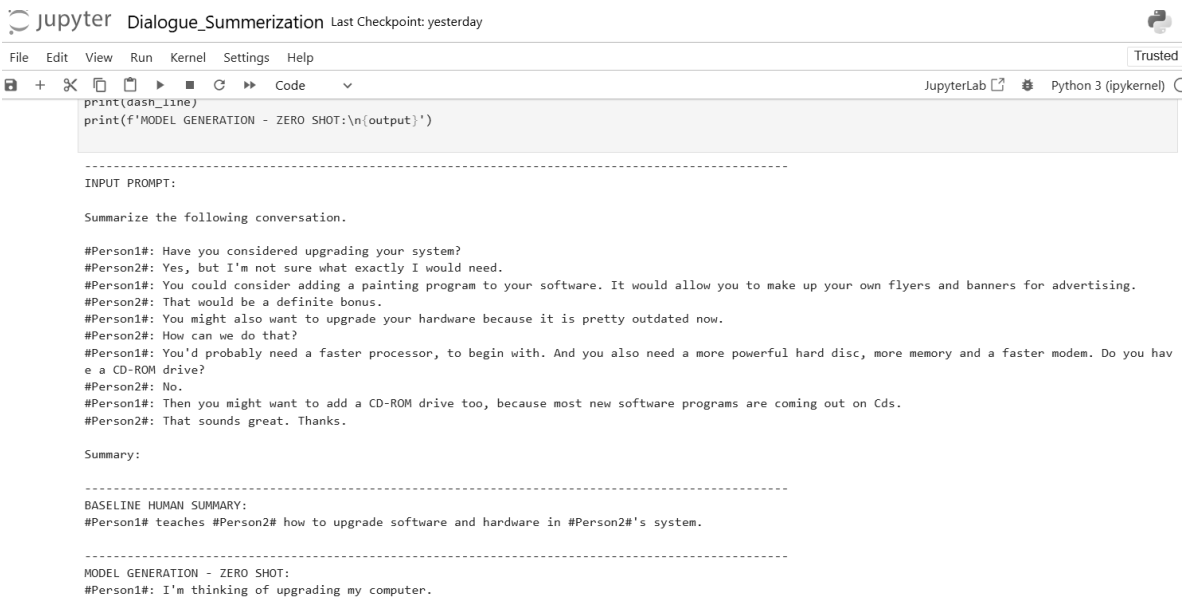


Figure 27. Zero-shot inference testing (Part 2).

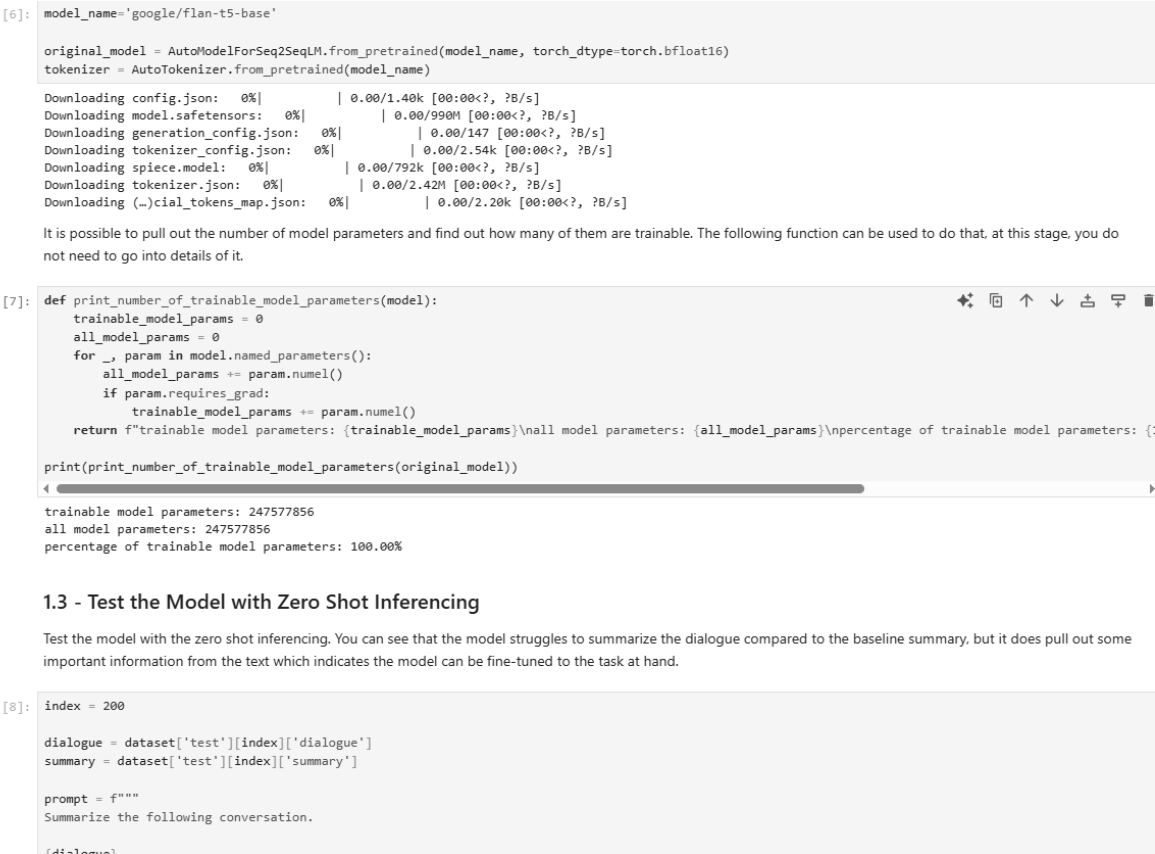


Figure 28. Testing model with zero-shot inference (Part 1).

```
[8]: index = 200

dialogue = dataset['test'][index]['dialogue']
summary = dataset['test'][index]['summary']

prompt = f"""
Summarize the following conversation.

{dialogue}

Summary:
"""

inputs = tokenizer(prompt, return_tensors='pt')
output = tokenizer.decode(
    original_model.generate(
        inputs["input_ids"],
        max_new_tokens=200,
    )[0],
    skip_special_tokens=True
)

dash_line = '-'.join(' ' for x in range(100))
print(dash_line)
print(f'INPUT PROMPT:\n{prompt}')
print(dash_line)
print(f'BASELINE HUMAN SUMMARY:\n{summary}\n')
print(dash_line)
print(f'MODEL GENERATION - ZERO SHOT:\n{output}')


-----
INPUT PROMPT:

Summarize the following conversation.

#Person1#: Have you considered upgrading your system?
#Person2#: Yes, but I'm not sure what exactly I would need.
#Person1#: You could consider adding a painting program to your software. It would allow you to make up your own flyers and banners for advertising.
#Person2#: That would be a definite bonus.
#Person1#: You might also want to upgrade your hardware because it is pretty outdated now.
#Person2#: How can we do that?
#Person1#: You'd probably need a faster processor, to begin with. And you also need a more powerful hard disc, more memory and a faster modem. Do you have a CD-ROM drive?
#Person2#: No.
#Person1#: Then you might want to add a CD-ROM drive too, because most new software programs are coming out on Cds.
#Person2#: That sounds great. Thanks.

Summary:
```

Figure 29. Testing model with zero-shot inference (Part 2).

 Jupyter Dialogue_Summerization Last Checkpoint: yesterday

File Edit View Run Kernel Settings Help

Trusted

JupyterLab Python 3 (ipykernel)

```
[46]: def make_prompt(example_indices_full, example_index_to_summarize):
    prompt = ''
    for index in example_indices_full:
        dialogue = dataset_test.iloc[index]['dialogue']
        summary = dataset_test.iloc[index]['summary']

        # The stop sequence '{summary}\n\n' is important for FLAN-T5. Other models may have their own preferred stop sequence.
        prompt += f"""
Dialogue:

{dialogue}

What was going on?
{summary}

"""

        dialogue = dataset_test.iloc[example_index_to_summarize]['dialogue']

        prompt += f"""
Dialogue:

{dialogue}

What was going on?
"""

    return prompt

[48]: example_indices_full = [40]
example_index_to_summarize = 200
```

Figure 30. One-shot inference code (Part 1).

Jupyter Dialogue_Summerization Last Checkpoint: yesterday

File Edit View Run Kernel Settings Help

Trusted

JupyterLab Python 3 (ipykernel)

```
[50]: summary = dataset_test.iloc[example_index_to_summarize]['summary']

inputs = tokenizer(one_shot_prompt, return_tensors='pt')
output = tokenizer.decode(
    model.generate(
        inputs["input_ids"],
        max_new_tokens=50,
    )[0],
    skip_special_tokens=True
)

print(dash_line)
print(f'BASELINE HUMAN SUMMARY:\n{summary}\n')
print(dash_line)
print(f'MODEL GENERATION - ONE SHOT:\n{output}')

-----
BASELINE HUMAN SUMMARY:
#Person1# teaches #Person2# how to upgrade software and hardware in #Person2#'s system.
-----
MODEL GENERATION - ONE SHOT:
#Person1 wants to upgrade his system. #Person2 wants to add a painting program to his software. #Person1 wants to add a CD-ROM drive.

[52]: example_indices_full = [40, 80, 120]
example_index_to_summarize = 200

few_shot_prompt = make_prompt(example_indices_full, example_index_to_summarize)

print(few_shot_prompt)

Dialogue:

#Person1#: What time is it, Tom?
```

Figure 31. One-shot inference code (Part 2).

Summarize Dialogue with One Shot and Few Shot Inference

One Shot Inference

```
[84]: def make_prompt(example_indices_full, example_index_to_summarize):
    prompt = ''
    for index in example_indices_full:
        dialogue = dataset_test.iloc[index]['dialogue']
        summary = dataset_test.iloc[index]['summary']

        # The stop sequence '{summary}\n\n' is important for FLAN-T5. Other models may have their own preferred stop sequence.
        prompt += f"""
Dialogue:

(dialogue)

What was going on?
(summary)

"""

        dialogue = dataset_test.iloc[example_index_to_summarize]['dialogue']

        prompt += f"""
Dialogue:

(dialogue)

What was going on?
"""

    return prompt
```

Figure 32. One-shot inference code (Part 3).

Jupyter

Dialogue_Summerization

Last Checkpoint: yesterday

File

Edit

View

Run

Kernel

Settings

Help

Trusted

+

✂

📄

📄

▶

■

🔄

⏩

Code

▼

JupyterLab

Python 3 (ipykernel)

[86]:

```
example_indices_full = [40]
example_index_to_summarize = 200

one_shot_prompt = make_prompt(example_indices_full, example_index_to_summarize)

print(one_shot_prompt)
```

Dialogue:

#Person1#: What time is it, Tom?
#Person2#: Just a minute. It's ten to nine by my watch.
#Person1#: Is it? I had no idea it was so late. I must be off now.
#Person2#: What's the hurry?
#Person1#: I must catch the nine-thirty train.
#Person2#: You've plenty of time yet. The railway station is very close. It won't take more than twenty minutes to get there.

What was going on?
#Person1# is in a hurry to catch a train. Tom tells #Person1# there is plenty of time.

Dialogue:

#Person1#: Have you considered upgrading your system?
#Person2#: Yes, but I'm not sure what exactly I would need.
#Person1#: You could consider adding a painting program to your software. It would allow you to make up your own flyers and banners for advertising.
#Person2#: That would be a definite bonus.
#Person1#: You might also want to upgrade your hardware because it is pretty outdated now.
#Person2#: How can we do that?
#Person1#: You'd probably need a faster processor, to begin with. And you also need a more powerful hard disc, more memory and a faster modem. Do you have a CD-ROM drive?
#Person2#: No.
#Person1#: Then you might want to add a CD-ROM drive too, because most new software programs are coming out on Cds.
#Person2#: That sounds great. Thanks.

What was going on?

Figure 33. One-shot inference code (Part 4).

Jupyter

Dialogue_Summerization

Last Checkpoint: yesterday

File

Edit

View

Run

Kernel

Settings

Help

Trusted

+

✂

📄

📄

▶

■

🔄

⏩

Code

▼

JupyterLab

Python 3 (ipykernel)

Dialogue:

#Person1#: Hello, I bought the pendant in your shop, just before.
#Person2#: Yes. Thank you very much.
#Person1#: Now I come back to the hotel and try to show it to my friend, the pendant is broken, I'm afraid.
#Person2#: Oh, is it?
#Person1#: Would you change it to a new one?
#Person2#: Yes, certainly. You have the receipt?
#Person1#: Yes, I do.
#Person2#: Then would you kindly come to our shop with the receipt by 10 o'clock? We will replace it.
#Person1#: Thank you so much.

What was going on?
#Person1# wants to change the broken pendant in #Person2#'s shop.

Dialogue:

#Person1#: Have you considered upgrading your system?
#Person2#: Yes, but I'm not sure what exactly I would need.
#Person1#: You could consider adding a painting program to your software. It would allow you to make up your own flyers and banners for advertising.
#Person2#: That would be a definite bonus.
#Person1#: You might also want to upgrade your hardware because it is pretty outdated now.
#Person2#: How can we do that?
#Person1#: You'd probably need a faster processor, to begin with. And you also need a more powerful hard disc, more memory and a faster modem. Do you have a CD-ROM drive?
#Person2#: No.
#Person1#: Then you might want to add a CD-ROM drive too, because most new software programs are coming out on Cds.
#Person2#: That sounds great. Thanks.

What was going on?

[54]:

```
summary = dataset_test.iloc[example_index_to_summarize]['summary']

inputs = tokenizer(few_shot_prompt, return_tensors='pt')
output = tokenizer.decode(
    model.generate(
```

Figure 34. One-shot inference input and output (Part 1).

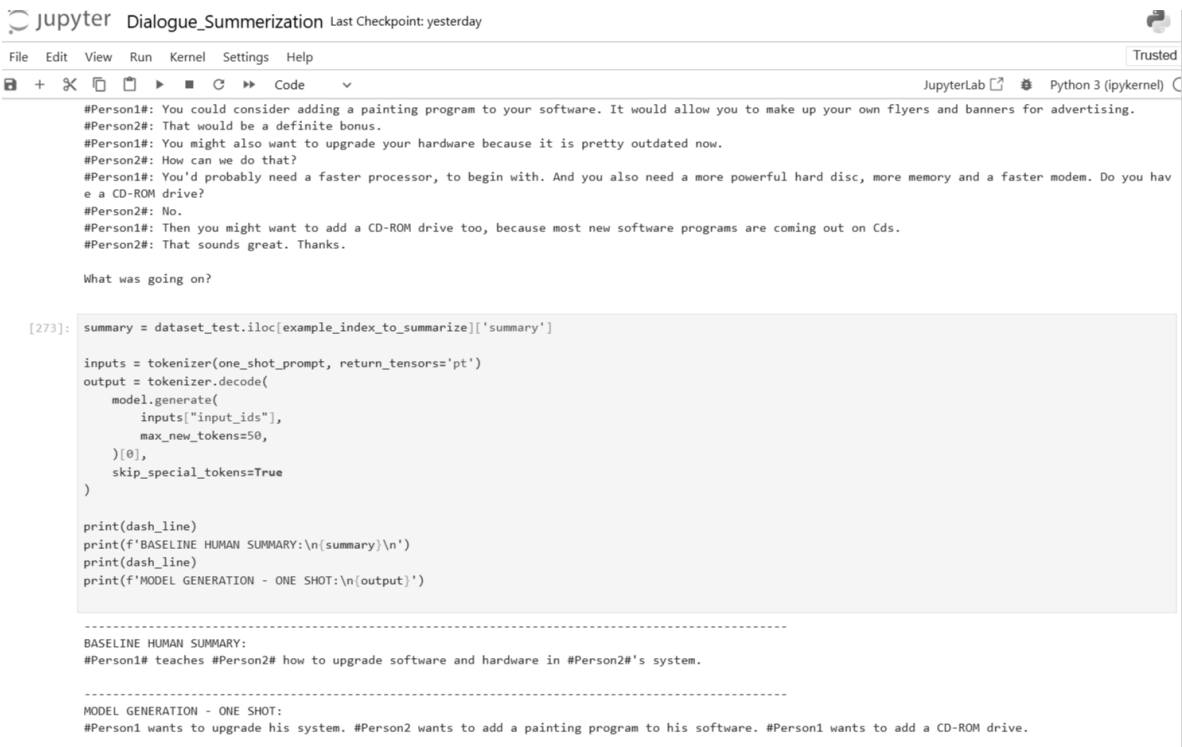


Figure 35. One-shot inference input and output (Part 2).

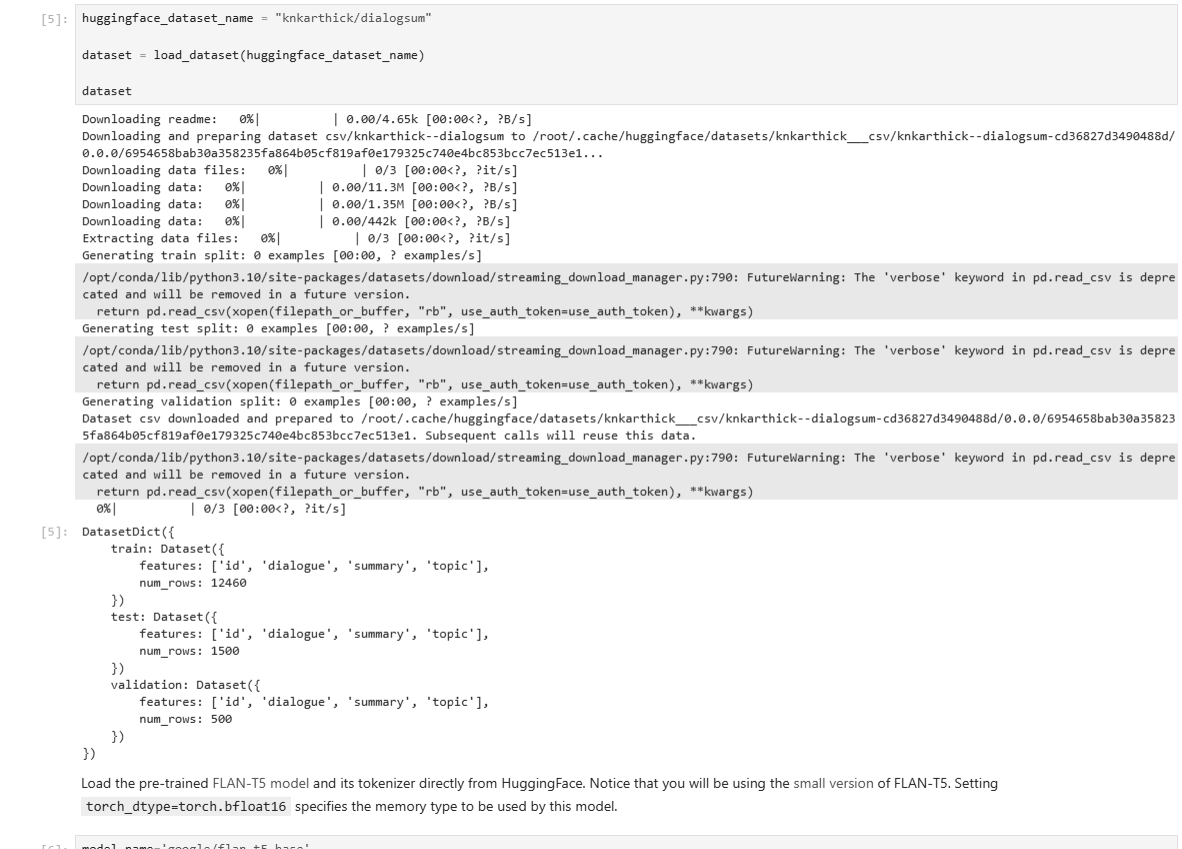


Figure 36. Loading the dataset for training.

```
[23]: from peft import LoraConfig, get_peft_model, TaskType

lora_config = LoraConfig(
    r=32, # Rank
    lora_alpha=32,
    target_modules=["q", "v"],
    lora_dropout=0.05,
    bias="none",
    task_type=TaskType.SEQ_2_SEQ_LM # FLAN-T5
)

Add LoRA adapter layers/parameters to the original LLM to be trained.

[24]: peft_model = get_peft_model(original_model,
                               lora_config)
print(print_number_of_trainable_model_parameters(peft_model))

trainable model parameters: 3538944
all model parameters: 251116800
percentage of trainable model parameters: 1.41%

3.2 - Train PEFT Adapter

Define training arguments and create Trainer instance.

[25]: output_dir = f'./peft-dialogue-summary-training-{str(int(time.time()))}'

peft_training_args = TrainingArguments(
    output_dir=output_dir,
    auto_find_batch_size=True,
    learning_rate=1e-3, # Higher learning rate than full fine-tuning.
    num_train_epochs=1,
    logging_steps=1,
    max_steps=1
)

peft_trainer = Trainer(
    model=peft_model,
    args=peft_training_args,
    train_dataset=tokenized_datasets["train"],
)
```

Now everything is ready to train the PEFT adapter and save the model.

Figure 37. LoRA and PEFT training implementation.

```
[26]: peft_trainer.train()

peft_model_path="./peft-dialogue-summary-checkpoint-local"

peft_trainer.model.save_pretrained(peft_model_path)
tokenizer.save_pretrained(peft_model_path)

/opt/conda/lib/python3.10/site-packages/transformers/optimization.py:391: FutureWarning: This implementation of AdamW is deprecated and will be removed in a future version. Use the PyTorch implementation torch.optim.AdamW instead, or set 'no_deprecation_warning=True' to disable this warning
warnings.warn(

[Progress bar] [1/1 00:00, Epoch 0/1]

Step   Training Loss
-----
1      51.000000

[27]: ('./peft-dialogue-summary-checkpoint-local/tokenizer_config.json',
      './peft-dialogue-summary-checkpoint-local/special_tokens_map.json',
      './peft-dialogue-summary-checkpoint-local/tokenizer.json')

That training was performed on a subset of data. To load a fully trained PEFT model, read a checkpoint of a PEFT model from S3.

[27]: !aws s3 cp --recursive s3://dlai-generative-ai/models/peft-dialogue-summary-checkpoint/ ./peft-dialogue-summary-checkpoint-from-s3/

huggingface/tokenizers: The current process just got forked, after parallelism has already been used. Disabling parallelism to avoid deadlocks...
To disable this warning, you can either:
  - Avoid using 'tokenizers' before the fork if possible
  - Explicitly set the environment variable TOKENIZERS_PARALLELISM=(true | false)
download: s3://dlai-generative-ai/models/peft-dialogue-summary-checkpoint/tokenizer_config.json to peft-dialogue-summary-checkpoint-from-s3/tokenizer_config.json
download: s3://dlai-generative-ai/models/peft-dialogue-summary-checkpoint/special_tokens_map.json to peft-dialogue-summary-checkpoint-from-s3/special_tokens_map.json
download: s3://dlai-generative-ai/models/peft-dialogue-summary-checkpoint/adapter_config.json to peft-dialogue-summary-checkpoint-from-s3/adapter_config.json
download: s3://dlai-generative-ai/models/peft-dialogue-summary-checkpoint/tokenizer.json to peft-dialogue-summary-checkpoint-from-s3/tokenizer.json
download: s3://dlai-generative-ai/models/peft-dialogue-summary-checkpoint/adapter_model.bin to peft-dialogue-summary-checkpoint-from-s3/adapter_model.bin

Check that the size of this model is much less than the original LLM:

[28]: !ls -al ./peft-dialogue-summary-checkpoint-from-s3/adapter_model.bin

huggingface/tokenizers: The current process just got forked, after parallelism has already been used. Disabling parallelism to avoid deadlocks...
To disable this warning, you can either:
  - Avoid using 'tokenizers' before the fork if possible
```

Figure 38. PEFT training implementation.

2 - Perform Full Fine-Tuning

2.1 - Preprocess the Dialog-Summary Dataset

You need to convert the dialog-summary (prompt-response) pairs into explicit instructions for the LLM. Prepend an instruction to the start of the dialog with `Summarize the following conversation` and to the start of the summary with `Summary` as follows:

Training prompt (dialogue):

```
Summarize the following conversation.

Chris: This is his part of the conversation.
Antje: This is her part of the conversation.

Summary:
```

Training response (summary):

```
Both Chris and Antje participated in the conversation.
```

Then preprocess the prompt-response dataset into tokens and pull out their `input_ids` (1 per token).

```
[9]: def tokenize_function(example):
    start_prompt = 'Summarize the following conversation.\n\n'
    end_prompt = '\n\nSummary: '
    prompt = [start_prompt + dialogue + end_prompt for dialogue in example["dialogue"]]
    example['input_ids'] = tokenizer(prompt, padding="max_length", truncation=True, return_tensors="pt").input_ids
    example['labels'] = tokenizer(example["summary"], padding="max_length", truncation=True, return_tensors="pt").input_ids

    return example

# The dataset actually contains 3 diff splits: train, validation, test.
# The tokenize_function code is handling all data across all splits in batches.
tokenized_datasets = dataset.map(tokenize_function, batched=True)
tokenized_datasets = tokenized_datasets.remove_columns(['id', 'topic', 'dialogue', 'summary',])

Map:   0%|          | 0/12460 [00:00<?, ? examples/s]
Map:   0%|          | 0/1500 [00:00<?, ? examples/s]
Map:   0%|          | 0/500 [00:00<?, ? examples/s]

To save some time in the lab, you will subsample the dataset:

[10]: tokenized_datasets = tokenized_datasets.filter(lambda example, index: index % 100 == 0, with_indices=True)
```

Figure 39. Performing full fine-tuning (Part 1).

```
[12]: output_dir = f'./dialogue-summary-training-{str(int(time.time()))}'

training_args = TrainingArguments(
    output_dir=output_dir,
    learning_rate=1e-5,
    num_train_epochs=1,
    weight_decay=0.01,
    logging_steps=1,
    max_steps=1
)

trainer = Trainer(
    model=original_model,
    args=training_args,
    train_dataset=tokenized_datasets['train'],
    eval_dataset=tokenized_datasets['validation']
)
```

Figure 40. Performing full fine-tuning (Part 2).

```
[25]: tokenizer = AutoTokenizer.from_pretrained(model_name, device_map="auto")

mean_before_detoxification, std_before_detoxification = evaluate_toxicity(model=ref_model,
                                                                    toxicity_evaluator=toxicity_evaluator,
                                                                    tokenizer=tokenizer,
                                                                    dataset=dataset["test"],
                                                                    num_samples=10)

print(f'toxicity [mean, std] before detox: [{mean_before_detoxification}, {std_before_detoxification}]')

11it [00:23, 2.13s/it]
toxicity [mean, std] before detox: [0.036252230225892905, 0.036904219097065025]
```

3 - Perform Fine-Tuning to Detoxify the Summaries

Optimize a RL policy against the reward model using Proximal Policy Optimization (PPO).

3.1 - Initialize PPOTrainer

For the PPOTrainer initialization, you will need a collator. Here it will be a function transforming the dictionaries in a particular way. You can define and test it:

```
[26]: def collator(data):
      return dict((key, [d[key] for d in data]) for key in data[0])

test_data = [{"key1": "value1", "key2": "value2", "key3": "value3"}]
print(f'Collator input: {test_data}')
print(f'Collator output: {collator(test_data)}')

Collator input: [{'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}]
Collator output: {'key1': ['value1'], 'key2': ['value2'], 'key3': ['value3']}
```

Set up the configuration parameters. Load the ppo_model and the tokenizer. You will also load a frozen version of the model ref_model. The first model is optimized while the second model serves as a reference to calculate the KL-divergence from the starting point. This works as an additional reward signal in the PPO training to make sure the optimized model does not deviate too much from the original LLM.

```
[27]: learning_rate=1.41e-5
max_ppo_epochs=1
mini_batch_size=4
batch_size=16

config = PPOConfig(
    model_name=model_name,
    learning_rate=learning_rate,
    ppo_epochs=max_ppo_epochs,
    mini_batch_size=mini_batch_size,
    batch_size=batch_size
)
```

Figure 41. PPO fine-tuning implementation.


```
[13]: ppo_model = AutoModelForSeq2SeqLMWithValueHead.from_pretrained(peft_model,
                                                                    torch_dtype=torch.bfloat16,
                                                                    is_trainable=True)

print(f'PPO model parameters to be updated (ValueHead + 769 params):\n(print_number_of_trainable_model_parameters(ppo_model))\n')
print(ppo_model.v_head)

Detected kernel version 4.14.336, which is below the recommended minimum of 5.5.0; this can cause the process to hang. It is recommended to upgrade the kernel to the minimum version or higher.
PPO model parameters to be updated (ValueHead + 769 params):

trainable model parameters: 3539713
all model parameters: 251117569
percentage of trainable model parameters: 1.41%

ValueHead(
  (dropout): Dropout(p=0.1, inplace=False)
  (summary): Linear(in_features=768, out_features=1, bias=True)
  (flatten): Flatten(start_dim=1, end_dim=-1)
)

During PPO, only a few parameters will be updated. Specifically, the parameters of the ValueHead. More information about this class of models can be found in the documentation. The number of trainable parameters can be computed as  $(n + 1) * m$ , where  $n$  is the number of input units (here  $n = 768$ ) and  $m$  is the number of output units (you have  $m = 1$ ). The  $+1$  term in the equation takes into account the bias term.

Now create a frozen copy of the PPO which will not be fine-tuned - a reference model. The reference model will represent the LLM before detoxification. None of the parameters of the reference model will be updated during PPO training. This is on purpose.
```

```
[14]: ref_model = create_reference_model(ppo_model)

print(f'Reference model parameters to be updated:\n(print_number_of_trainable_model_parameters(ref_model))\n')

Reference model parameters to be updated:

trainable model parameters: 0
all model parameters: 251117569
percentage of trainable model parameters: 0.00%

Everything is set. It is time to prepare the reward model!
```

2.2 - Prepare Reward Model

Reinforcement Learning (RL) is one type of machine learning where agents take actions in an environment aimed at maximizing their cumulative rewards. The agent's behavior is defined by the **policy**. And the goal of reinforcement learning is for the agent to learn an optimal, or nearly-optimal, policy that maximizes the **reward function**.

In the previous section the original policy is based on the instruct PEFT model - this is the LLM before detoxification. Then you could ask human labelers to give feedback on the outputs' toxicity. However, it can be expensive to use them for the entire fine-tuning process. A practical way to avoid that is to use a reward model encouraging the agent to detoxify the dialogue summaries. The intuitive approach would be to do some form of sentiment analysis across two classes (`nothate` and `hate`) and give a higher reward if there is higher a chance of getting class `nothate` as an output.

Figure 42. PPO model implementation.

```
[17]: toxic_text = "#Person 1# tells Tommy that the movie was terrible, dumb and stupid."

toxicity_input_ids = toxicity_tokenizer(toxic_text, return_tensors="pt").input_ids

logits = toxicity_model(toxicity_input_ids).logits
print(f'Logits [not hate, hate]: {logits.tolist()[0]}')

# Print the probabilities for [not hate, hate]
probabilities = logits.softmax(dim=-1).tolist()[0]
print(f'probabilities [not hate, hate]: {probabilities}')

# Get the Logits for "not hate" - this is the reward!
nothate_reward = (logits[:, not_hate_index]).tolist()
print(f'reward (low): {nothate_reward}')

logits [not hate, hate]: [-0.6921188831329346, 0.3722729980945587]
probabilities [not hate, hate]: [0.25647106766700745, 0.7435289621353149]
reward (low): [-0.6921188831329346]

Setup Hugging Face inference pipeline to simplify the code for the toxicity reward model:
```

```
[18]: device = 0 if torch.cuda.is_available() else "cpu"

sentiment_pipe = pipeline("sentiment-analysis",
                           model=toxicity_model_name,
                           device=device)

reward_logits_kwargs = {
    "top_k": None, # Return all scores.
    "function_to_apply": "none", # Set to "none" to retrieve raw Logits.
    "batch_size": 16
}

reward_probabilities_kwargs = {
    "top_k": None, # Return all scores.
    "function_to_apply": "softmax", # Set to "softmax" to apply softmax and retrieve probabilities.
    "batch_size": 16
}

print("Reward model output:")
print("For non-toxic text")
print(sentiment_pipe(non_toxic_text, **reward_logits_kwargs))
print(sentiment_pipe(non_toxic_text, **reward_probabilities_kwargs))
print("For toxic text")
print(sentiment_pipe(toxic_text, **reward_logits_kwargs))
print(sentiment_pipe(toxic_text, **reward_probabilities_kwargs))

Reward model output:
For non-toxic text
[{'label': 'nothate', 'score': 3.114100694656372}, {'label': 'hate', 'score': -2.4896175861358643}]
[{'label': 'nothate', 'score': 0.9963293671607971}, {'label': 'hate', 'score': 0.003670616541057825}]
For toxic text
[{'label': 'hate', 'score': 0.3722729980945587}, {'label': 'nothate', 'score': -0.6921188831329346}]
[{'label': 'hate', 'score': 0.7435289621353149}, {'label': 'nothate', 'score': 0.25647106766700745}]
```

Figure 43. Reward model output for PPO fine-tuning.

2.3 - Evaluate the Model Qualitatively (Human Evaluation)

As with many GenAI applications, a qualitative approach where you ask yourself the question "Is my model behaving the way it is supposed to?" is usually a good starting point. In the example below (the same one we started this notebook with), you can see how the fine-tuned model is able to create a reasonable summary of the dialogue compared to the original inability to understand what is being asked of the model.

```
index = 200
dialogue = dataset['test'][index]['dialogue']
human_baseline_summary = dataset['test'][index]['summary']

prompt = f"""
Summarize the following conversation.

{dialogue}

Summary:
"""

input_ids = tokenizer(prompt, return_tensors="pt").input_ids

original_model_outputs = original_model.generate(input_ids=input_ids, generation_config=GenerationConfig(max_new_tokens=200, num_beams=1))
original_model_text_output = tokenizer.decode(original_model_outputs[0], skip_special_tokens=True)

instruct_model_outputs = instruct_model.generate(input_ids=input_ids, generation_config=GenerationConfig(max_new_tokens=200, num_beams=1))
instruct_model_text_output = tokenizer.decode(instruct_model_outputs[0], skip_special_tokens=True)

print(dash_line)
print(f'BASELINE HUMAN SUMMARY:\n{human_baseline_summary}')
print(dash_line)
print(f'ORIGINAL MODEL:\n{original_model_text_output}')
print(dash_line)
print(f'INSTRUCT MODEL:\n{instruct_model_text_output}')
```

BASELINE HUMAN SUMMARY:
#Person1# teaches #Person2# how to upgrade software and hardware in #Person2#'s system.

ORIGINAL MODEL:
#Person1#: You'd like to upgrade your computer. #Person2: You'd like to upgrade your computer.

INSTRUCT MODEL:
#Person1# suggests #Person2# upgrading #Person2#'s system, hardware, and CD-ROM drive. #Person2# thinks it's great.

Figure 44. Qualitative evaluation of the model (Part 1).

```
[29]: from peft import PeftModel, PeftConfig

peft_model_base = AutoModelForSeq2SeqLM.from_pretrained("google/flan-t5-base", torch_dtype=torch.bfloat16)
tokenizer = AutoTokenizer.from_pretrained("google/flan-t5-base")

peft_model = PeftModel.from_pretrained(peft_model_base,
                                      './peft-dialogue-summary-checkpoint-from-s3/',
                                      torch_dtype=torch.bfloat16,
                                      is_trainable=False)

The number of trainable parameters will be 0 due to is_trainable=False setting:
```

```
[30]: print(print_number_of_trainable_model_parameters(peft_model))

trainable model parameters: 0
all model parameters: 251116800
percentage of trainable model parameters: 0.00%
```

3.3 - Evaluate the Model Qualitatively (Human Evaluation)

Make inferences for the same example as in sections 1.3 and 2.3, with the original model, fully fine-tuned and PEFT model.

```
[ ]: index = 200
dialogue = dataset['test'][index]['dialogue']
baseline_human_summary = dataset['test'][index]['summary']

prompt = f"""
Summarize the following conversation.

{dialogue}

Summary: """

input_ids = tokenizer(prompt, return_tensors="pt").input_ids

original_model_outputs = original_model.generate(input_ids=input_ids, generation_config=GenerationConfig(max_new_tokens=200, num_beams=1))
original_model_text_output = tokenizer.decode(original_model_outputs[0], skip_special_tokens=True)

instruct_model_outputs = instruct_model.generate(input_ids=input_ids, generation_config=GenerationConfig(max_new_tokens=200, num_beams=1))
instruct_model_text_output = tokenizer.decode(instruct_model_outputs[0], skip_special_tokens=True)

peft_model_outputs = peft_model.generate(input_ids=input_ids, generation_config=GenerationConfig(max_new_tokens=200, num_beams=1))
peft_model_text_output = tokenizer.decode(peft_model_outputs[0], skip_special_tokens=True)

print(dash_line)
```

Figure 45. Qualitative evaluation of the model (Part 2).

2.4 - Evaluate the Model Quantitatively (with ROUGE Metric)

The ROUGE metric helps quantify the validity of summarizations produced by models. It compares summarizations to a "baseline" summary which is usually created by a human. While not perfect, it does indicate the overall increase in summarization effectiveness that we have accomplished by fine-tuning.

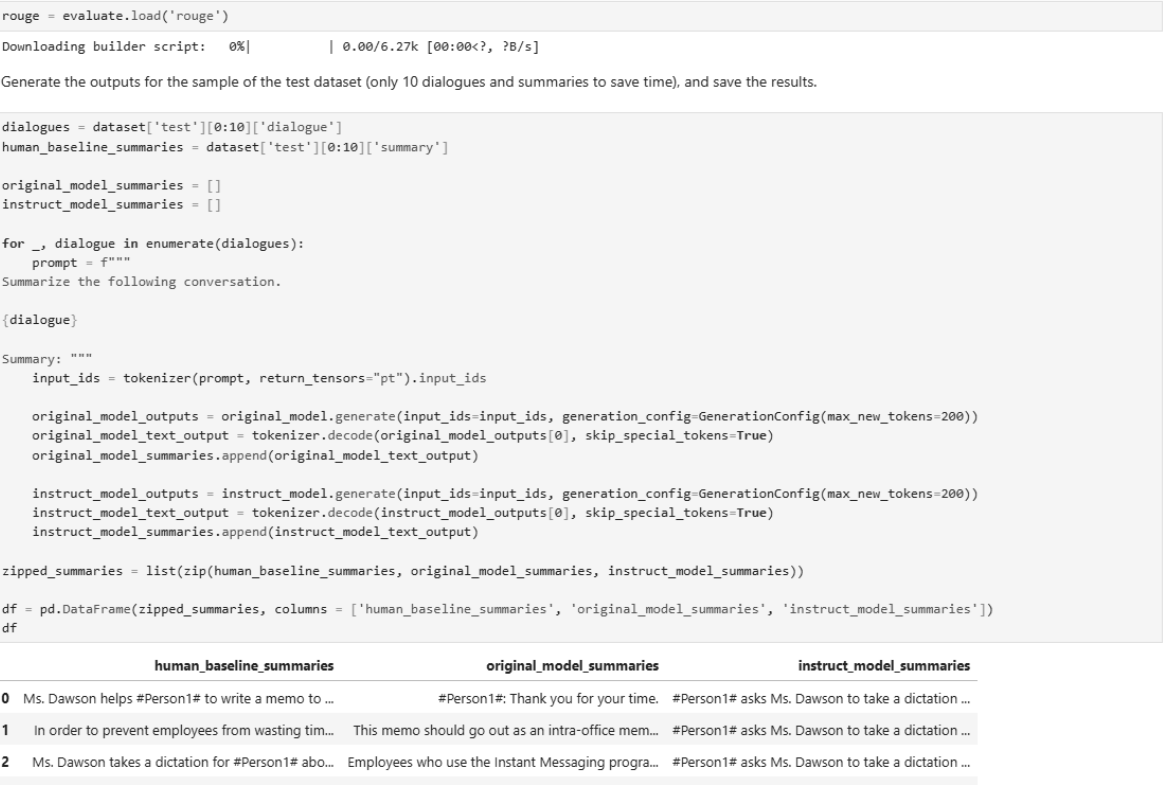


Figure 46. Quantitative evaluation of the model (Part 1).

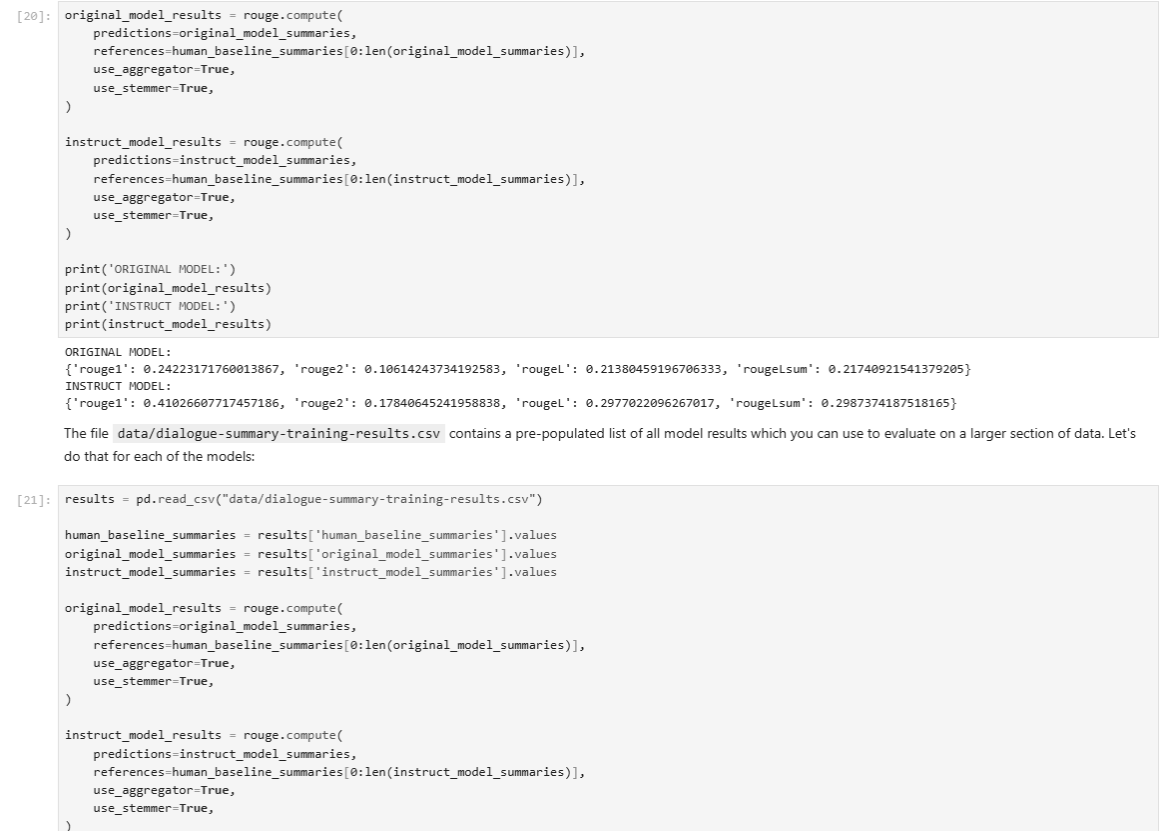


Figure 47. Quantitative evaluation of the model (Part 2).

```
[21]: results = pd.read_csv("data/dialogue-summary-training-results.csv")

human_baseline_summaries = results['human_baseline_summaries'].values
original_model_summaries = results['original_model_summaries'].values
instruct_model_summaries = results['instruct_model_summaries'].values

original_model_results = rouge.compute(
    predictions=original_model_summaries,
    references=human_baseline_summaries[0:len(original_model_summaries)],
    use_aggregator=True,
    use_stemmer=True,
)

instruct_model_results = rouge.compute(
    predictions=instruct_model_summaries,
    references=human_baseline_summaries[0:len(instruct_model_summaries)],
    use_aggregator=True,
    use_stemmer=True,
)

print('ORIGINAL MODEL:')
print(original_model_results)
print('INSTRUCT MODEL:')
print(instruct_model_results)

ORIGINAL MODEL:
{'rouge1': 0.2334158581572823, 'rouge2': 0.07603964187010573, 'rougeL': 0.20145520923859048, 'rougeLsum': 0.20145899339006135}
INSTRUCT MODEL:
{'rouge1': 0.42161291557556113, 'rouge2': 0.18035380596301792, 'rougeL': 0.3384439349963909, 'rougeLsum': 0.33835653595561666}

The results show substantial improvement in all ROUGE metrics:
```

```
[22]: print("Absolute percentage improvement of INSTRUCT MODEL over ORIGINAL MODEL")

improvement = (np.array(list(instruct_model_results.values())) - np.array(list(original_model_results.values())))
for key, value in zip(instruct_model_results.keys(), improvement):
    print(f'{key}: {value*100:.2f}%')

Absolute percentage improvement of INSTRUCT MODEL over ORIGINAL MODEL
rouge1: 18.82%
rouge2: 10.43%
rougeL: 13.70%
rougeLsum: 13.69%
```

Figure 48. Quantitative evaluation of the model (Part 3).

```
[ ]: dialogues = dataset['test'][0:10]['dialogue']
human_baseline_summaries = dataset['test'][0:10]['summary']

original_model_summaries = []
instruct_model_summaries = []
peft_model_summaries = []

for idx, dialogue in enumerate(dialogues):
    prompt = f"""
    Summarize the following conversation.

    {dialogue}

    Summary: """

    input_ids = tokenizer(prompt, return_tensors="pt").input_ids

    human_baseline_text_output = human_baseline_summaries[idx]

    original_model_outputs = original_model.generate(input_ids=input_ids, generation_config=GenerationConfig(max_new_tokens=200))
    original_model_text_output = tokenizer.decode(original_model_outputs[0], skip_special_tokens=True)

    instruct_model_outputs = instruct_model.generate(input_ids=input_ids, generation_config=GenerationConfig(max_new_tokens=200))
    instruct_model_text_output = tokenizer.decode(instruct_model_outputs[0], skip_special_tokens=True)

    peft_model_outputs = peft_model.generate(input_ids=input_ids, generation_config=GenerationConfig(max_new_tokens=200))
    peft_model_text_output = tokenizer.decode(peft_model_outputs[0], skip_special_tokens=True)

    original_model_summaries.append(original_model_text_output)
    instruct_model_summaries.append(instruct_model_text_output)
    peft_model_summaries.append(peft_model_text_output)

zipped_summaries = list(zip(human_baseline_summaries, original_model_summaries, instruct_model_summaries, peft_model_summaries))

df = pd.DataFrame(zipped_summaries, columns = ['human_baseline_summaries', 'original_model_summaries', 'instruct_model_summaries', 'peft_model_summaries'])
df

Compute ROUGE score for this subset of the data.
```

```
[ ]: rouge = evaluate.load('rouge')

original_model_results = rouge.compute(
    predictions=original_model_summaries,
    references=human_baseline_summaries[0:len(original_model_summaries)],
    use_aggregator=True,
    use_stemmer=True,
```

Figure 49. Quantitative evaluation of the model (Part 4).

Compute ROUGE score for this subset of the data.

```
]: rouge = evaluate.load('rouge')

original_model_results = rouge.compute(
    predictions=original_model_summaries,
    references=human_baseline_summaries[0:len(original_model_summaries)],
    use_aggregator=True,
    use_stemmer=True,
)

instruct_model_results = rouge.compute(
    predictions=instruct_model_summaries,
    references=human_baseline_summaries[0:len(instruct_model_summaries)],
    use_aggregator=True,
    use_stemmer=True,
)

peft_model_results = rouge.compute(
    predictions=peft_model_summaries,
    references=human_baseline_summaries[0:len(peft_model_summaries)],
    use_aggregator=True,
    use_stemmer=True,
)

print('ORIGINAL MODEL:')
print(original_model_results)
print('INSTRUCT MODEL:')
print(instruct_model_results)
print('PEFT MODEL:')
print(peft_model_results)
```

Notice, that PEFT model results are not too bad, while the training process was much easier!

You already computed ROUGE score on the full dataset, after loading the results from the `data/dialogue-summary-training-results.csv` file. Load the values for the PEFT model now and check its performance compared to other models.

```
]: human_baseline_summaries = results['human_baseline_summaries'].values
original_model_summaries = results['original_model_summaries'].values
instruct_model_summaries = results['instruct_model_summaries'].values
peft_model_summaries = results['peft_model_summaries'].values

original_model_results = rouge.compute(
    predictions=original_model_summaries,
    references=human_baseline_summaries[0:len(original_model_summaries)],
    use_aggregator=True,
```

Figure 50. Quantitative evaluation of the model (Part 5).

```
[ ]: human_baseline_summaries = results['human_baseline_summaries'].values
original_model_summaries = results['original_model_summaries'].values
instruct_model_summaries = results['instruct_model_summaries'].values
peft_model_summaries = results['peft_model_summaries'].values

original_model_results = rouge.compute(
    predictions=original_model_summaries,
    references=human_baseline_summaries[0:len(original_model_summaries)],
    use_aggregator=True,
    use_stemmer=True,
)

instruct_model_results = rouge.compute(
    predictions=instruct_model_summaries,
    references=human_baseline_summaries[0:len(instruct_model_summaries)],
    use_aggregator=True,
    use_stemmer=True,
)

peft_model_results = rouge.compute(
    predictions=peft_model_summaries,
    references=human_baseline_summaries[0:len(peft_model_summaries)],
    use_aggregator=True,
    use_stemmer=True,
)

print('ORIGINAL MODEL:')
print(original_model_results)
print('INSTRUCT MODEL:')
print(instruct_model_results)
print('PEFT MODEL:')
print(peft_model_results)
```

The results show less of an improvement over full fine-tuning, but the benefits of PEFT typically outweigh the slightly-lower performance metrics.

Calculate the improvement of PEFT over the original model:

```
[ ]: print("Absolute percentage improvement of PEFT MODEL over ORIGINAL MODEL")

improvement = (np.array(list(peft_model_results.values())) - np.array(list(original_model_results.values())))/
for key, value in zip(peft_model_results.keys(), improvement):
    print(f'{key}: {value*100:.2f}%')
```

Now calculate the improvement of PEFT over a full fine-tuned model:

Figure 51. Quantitative evaluation of the model (Part 6).

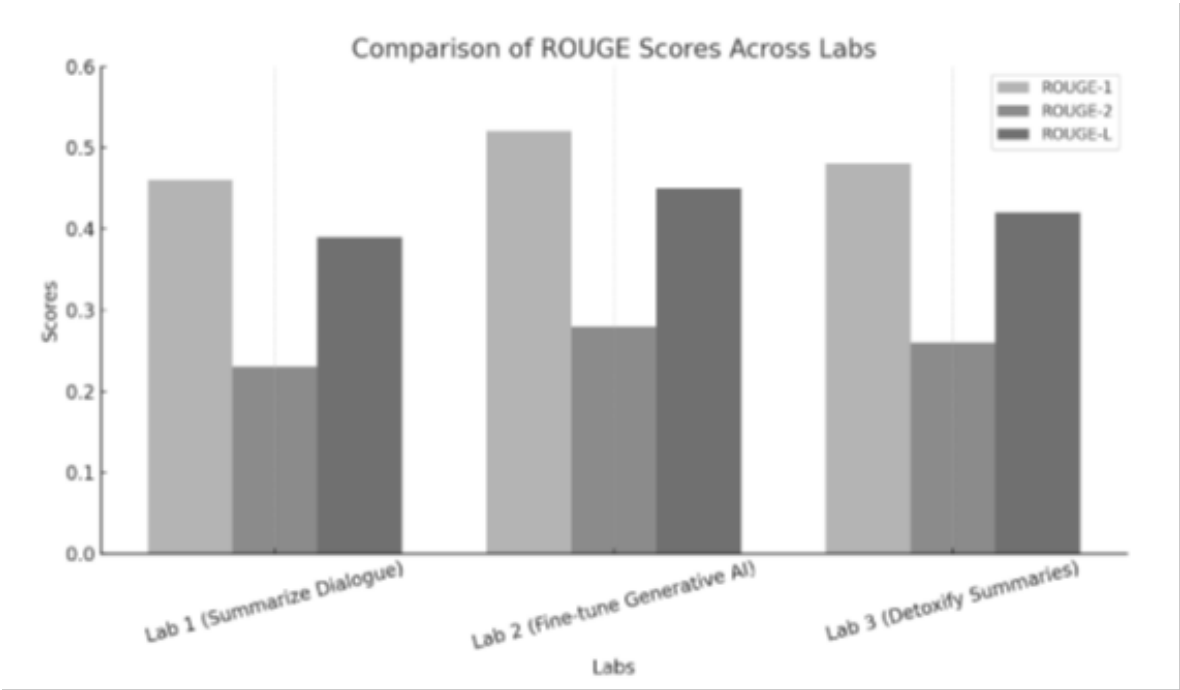


Figure 52. Comparison of Rouge Scores Across Labs.

References

1. Yao, J. The Application of Generative Artificial Intelligence in Education: Potential, Challenges, and Strategies. *SHS Web of Conferences* **2024**, 200, 02008. <https://doi.org/10.1051/shsconf/202420002008>.

2. Lee, D.; et al. The impact of generative AI on higher education learning and teaching: A study of educators' perspectives. *Computers and Education: Artificial Intelligence* **2024**, 6, 100221. <https://doi.org/10.1016/j.caeai.2024.100221>.

3. Denny, P.; et al. Generative AI for Education (GAIED): Advances, Opportunities, and Challenges. *arXiv preprint arXiv:2402.01580* **2024**.

4. Maity, S.; et al. Generative AI and Its Impact on Personalized Intelligent Tutoring Systems. *arXiv preprint arXiv:2410.10650* **2024**.

5. Balderas, L.; et al. Generative AI in Higher Education: Teachers' and Students' Perspectives on Support, Replacement, and Digital Literacy. *Education Sciences* **2024**, 15, 396. <https://doi.org/10.3390/educsci15040396>.

6. Haroud, S.; Saqri, N. Generative AI in Higher Education: Teachers' and Students' Perspectives on Support, Replacement, and Digital Literacy. *Education Sciences* **2025**, 15, 396. <https://doi.org/10.3390/educsci15040396>.

7. Kingma, D.P.; Welling, M. Auto-Encoding Variational Bayes. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2014.

8. Goodfellow, I.J.; Pouget-Abadie, J.; Mirza, M.; Xu, B.; Warde-Farley, D.; Ozair, S.; Courville, A.; Bengio, Y. Generative Adversarial Nets. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, 2014, Vol. 27.

9. Rezende, D.J.; Mohamed, S. Variational Inference with Normalizing Flows. In *Proceedings of the Proceedings of the 32nd International Conference on Machine Learning (ICML)*, Lille, France, 2015; pp. 1530–1538.

10. Ho, J.; Jain, A.; Abbeel, P. Denoising Diffusion Probabilistic Models. In *Proceedings of the Advances in Neural Information Processing Systems*, 2020, Vol. 33, pp. 6840–6851.

11. van den Oord, A.; Dieleman, S.; Zen, H.; et al. WaveNet: A Generative Model for Raw Audio. *arXiv preprint arXiv:1609.03499* **2016**.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.