Advanced Generative Multi-turn Chatbot Design and Implementation

Mohamed Niaz .M, Rishabh Malik, Aleena Varghese

Group 3

Applications of Artificial Intelligence, University of SanDiego

AAI-520: Natural Language Processing and GenAI

Haisav Chokshi

October 21, 2024

Table of Contents

Advanced Multi-turn Conversation Chatbot Design & Implementation

This project developed a chatbot with multi-turn conversational ability, capable of adapting to context and handling a variety of topics. The chatbot is based on the T5-base text-to-text model (Raffel et al., 2019) with FLAN-T5 checkpoint, which is finetuned for language-based instructions (Chung et al., 2022) that suits well for a chatbot. The model was trained using the Cornell Movie Dialogue corpus (*Cornell Movie-Dialog Corpus*, 2018), providing it with a rich dataset for conversational training.

The trained chatbot model is accessible through a text-based interface and web interface, allowing users to engage with the bot in multiple ways. The chatbot engine retains the history of the conversation for programmable number of turns to preserve the context as well as to adapt the context over the turn length. Additionally, the engine includes configurable controls to adjust the randomness of the responses enhancing the diversity and creativity of the dialogue.

The chatbot has been validated with many different conversation scenarios to test its ability to retain context and adapt to varying conversational topics. These test the chatbot's effectiveness in maintaining coherent conversation over extended interactions. The developed codebase along with the trained model is the GitHub location https://github.com/RishabhDE/MultiTurnChatbotEngine

In addition, we've fine-tuned our dataset on GPT-4o using the OpenAI API to achieve even better results. This enhancement leverages the advanced capabilities of GPT-4o, enabling the chatbot to deliver more contextually relevant and nuanced responses. Fine-tuning with OpenAI's API has further optimized the model's ability to handle complex conversational threads, improving its adaptability to diverse topics and enhancing the overall user experience.

Data Preprocessing

The Cornell movie dialogue corpus has 220,579 conversational exchanges having total 304,446 utterances (*Cornell Movie-Dialog Corpus*, 2018). Being a movie dialogue corpus, the dataset contains a significant amount of colloquial content, which required extensive cleaning. Contractions such as "we've" are expanded, and punctuation anomalies such as, hyphens, multiple dots and consecutive space were removed to standardize the text.

The dialogue word count distribution of the cleaned-up dialogue revealed that the average word count per dialogue was 11.08 words, with nearly all the dialogues containing less than 64 words. The response quality of the chatbot largely depends on the number of words in the response, especially when monosyllabic responses such as 'yes', 'no' and 'ok' are used sparingly. The most frequently occurring words in the cleaned dataset was inspected to find for such words and found to be fine for further processing.

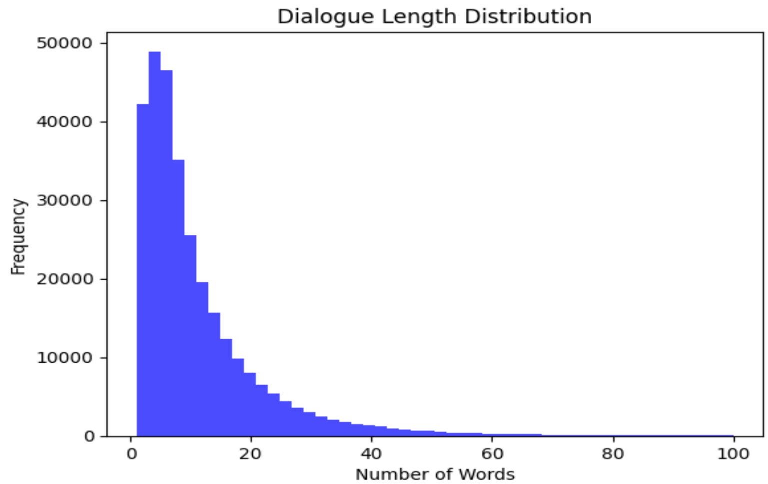


Figure 1. Dialogue length Distribution (number of words)

The cleaned-up dialogues were paired to create question answer (QA) pairs. To ensure response quality, the QA dataset was refined by filtering out pairs with low keyword commonality between the question and answer. Additionally, QA pairs with response significantly shorter than the question, or monosyllabic in nature were removed from the dataset. The refined dataset had 145,657 QA pairs.

The QA pair was further refined by checking the cosine semantic similarity between the QA pairs using sentence embeddings generated from a pretrained model. The cosine similarity distribution is plotted to visualize the suitable threshold, revealing that that most of the conversation exhibit 20% or less semantic similarity in this dataset. To maintain sufficient training dataset volume the similarity threshold was set to 5%, resulting in 135,992 refined QA pairs, which were used to train the chatbot model.

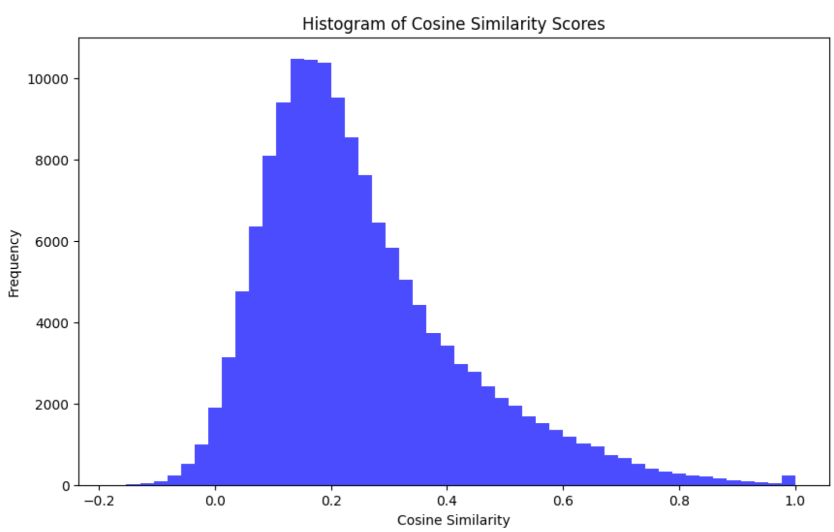


Figure 2. Histogram of QA Pair Cosine Similarity Scores

Model Selection

This project uses the T5 transformer based pre-trained model, which is designed for text-to-text transformation (Raffel et al., 2019), making it ideal for the QA model of the chatbot. Specifically, we utilize the FLAN-T5 checkpoint, which is fine-tuned for instruction-based tasks (Chung et al., 2022) such as question answering and conversational interactions. FLAN-T5 has been trained on a variety of NLP tasks, making it robust to handle diverse topics and conversational context shifts.

The T5-base model was selected due to its versatility compared to the T5-small model, while being compact relative to the T5-large model, which strikes better balance between performance and computation efficiency.

Models such as GPT-3 and GPT-4 were not considered for this project due to the significant computational resources required the limited fine-tuning capabilities through the API. Proof-of-concept experiments with other transformer models such as DialoGPT and BERT confirmed the relative merits of FLAN-T5, making it the preferred choice for this project.

Model Training

A custom PyTorch dataset was developed for tokenizing the QA dialogue pairs with 64-token padded truncation, returning input token IDs, input attention masks and output (label) token IDs. To facilitate efficient training and evaluation, a data loader function was implemented that loads batches of shuffled into the model. The batch size and number of workers were optimized based on the capabilities of Nvidia A100 GPU used for training.

The training can either start from scratch or resume from where it stopped in a previous run, enabling incremental training. Intermediate checkpoints are saved after every 500 steps and the final model is saved after training. The batch size, learning rate and gradient accumulation steps are tuned to leverage the A100 GPU capabilities. The learning rate schedule includes a warmup phase during which the learning rate increases and then slowly decays while training progresses, which ensures training stability. The code also adapts the learning rate if the batch size is changed during training. When training resumes, the optimizer and scheduler states are restored to ensure smooth continuation of the training progress.

Model Evaluation and Results

Traditional metrics such as precision, recall and F1 Score are not applicable to the generative chatbot task. Metrics such as BLEU and ROUGE are based on exact word overlaps, commonly used for machine translation and summarization. However, they are inadequate to assess generative conversational models, as they fail to capture open ended nature of dialogue and the huge variability in potential responses.

Perplexity is a more suitable metric for generative models, which measures how well the model predicts a given sequence of words, quantifying the uncertainty of the model when encountering new data (Jurafsky et al., 2024). Perplexity is calculated as the exponential of the evaluation loss after the model is trained. Given the limited volume of the training data, the perplexity score is very high. Training with the same dataset will lead to overfitting.

One approach to mitigate the high perplexity is to generate synthetic data from the original dataset to augment the training, which requires substantial computing power. Another approach is to evaluate the chatbot based on the conversation quality and context coherence, which reflects real world performance. Annexure -C provides a few sample conversations based on which the model conversation quality and context coherence are analyzed. Based on the evaluation, the chatbot could maintain the context very well and able to adapt the context based on the configured conversation length. However, if the dialogue goes beyond the scope of the chatbot, it repeats previous dialogue.

Chatbot Interface

The chatbot interface is developed with both textual and GUI interface with a common underlying response generator, which maintains context of the multi-turn conversation over configurable depth. The user text is added to the context history, truncated to the right depth, tokenized and response generated from the model based on set of conversation control parameters.

The key conversation control parameters are temperature controlling the randomness of the model's response, num_beams controlling the number of explored beams for potential response, top_k defining number of top likely tokens to be sampled, top_p defining cumulative probability for choosing tokens based on Nucleus sampling and rep_penality which penalizes for repeating a token often.

State-of-the-Art GPT-4.0 mini-Model

In an effort to explore the next generation of transformer models, the GPT-4.0 mini model was trained with the Cornell Movie Dialogue corpus using the OpenAI API. Training via the API allowed us to seamlessly integrate OpenAI's advanced language capabilities, enabling precise fine-tuning that enhanced the chatbot’s conversational proficiency and contextual awareness.

Additionally, training the GPT-4.0 mini model on the Cornell dataset enriched the bot’s ability to handle dynamic, multi-turn conversations. The API-driven fine-tuning refined the model's responses, improving its versatility in managing diverse topics while maintaining coherent, contextually relevant dialogue over extended interactions. These improvements highlight the potential of GPT-4.0 in advancing AI-driven conversational systems. (Trained model link: https://group3-usd.streamlit.app/)

Code Documentation

The codebase for the Flan-T5 Chatbot model and GPT-4.0 model are detailed with adequate explanation in Annexure-A and Annexure-B for reference. The developed code is available in the GitHub.

Challenges Faced

The selection of suitable model architecture has been a complex process. Initially the T5 model was chosen due to the text-to-text nature of the chatbot. However, the T5-small yielded moderate performance. Other models, such as GPT-2 and DialoGPT, were explored, but could not offer satisfactory performances. As a result, the Flan-T5 checkpoint of T5 model, which is fine-tuned with language-based instruction, was selected for this project.

To improve conversation quality, the T5-base model was adopted later instead of T5-small, which introduced computation resource challenges. Using data exploration, token length has been optimized to 64, and the batch size and number of workers were tuned to efficiently use the A100 GPU, resulting in a satisfactory training speed.

The limited quantity of qualified QA pairs was another challenge in achieving satisfactory performance with the T5-base model. The memory availability with the available computation platforms was inadequate to try with the T5-large model. Hence, GPT-4.0 mini was tried out, which offered the OpenAI based API methodology to train the state-of-the-art model. The conversation quality was reasonably good.

Conclusion

In conclusion, this project showcases the successful design and implementation of an advanced multi-turn conversation chatbot that utilizes the T5-base model with the FLAN-T5 checkpoint, fine-tuned on the Cornell Movie Dialogue corpus. The chatbot demonstrates a strong ability to engage in contextually relevant dialogues across a wide range of topics. The integration of the GPT-4.0 mini model, facilitated through the OpenAI API, further enhances its performance, enabling nuanced responses and improved adaptability to complex conversational dynamics.

Despite facing challenges in model selection and resource limitations, effective data preprocessing and QA pair refinement helped address issues related to response quality and coherence. The results highlight the chatbot's potential for real-world applications and provide valuable insights for further development in generative AI-driven conversational systems, paving the way for future improvements in AI technologies.

References

Chung, H. W., Hou, L., Longpre, S., Zoph, B., Tay, Y., Fedus, W., Li, E., Wang, X., Dehghani, M., Brahma, S., Webson, A., Gu, S. S., Dai, Z., Suzgun, M., Chen, X., Chowdhery, A., Narang, S., Mishra, G., Yu, A., Wei, J. (2022). Scaling Instruction-Finetuned Language Models. arXiv (Cornell University). https://doi.org/10.48550/arxiv.2210.11416

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2019). Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. arXiv (Cornell University). https://doi.org/10.48550/arxiv.1910.10683

Cornell Movie-Dialog Corpus. (2018, March 28). Kaggle. https://www.kaggle.com/datasets/rajathmc/cornell-moviedialog-corpus

Jurafsky, D., & Martin, J. H. (2024). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models* (3rd ed.). https://web.stanford.edu/~jurafsky/slp3/

RishabhDE. (n.d.). *GitHub - RishabhDE/MultiModalChatbotEngine*. GitHub. https://github.com/RishabhDE/MultiModalChatbotEngine

Appendix A – Code Documentation – FLAN-T5 Chatbot

Preparation of Compute Environment

Installation and Import of Libraries Needed

- This section of the code installs additional libraries needed in the execution environment.

```
# Sentence Transformers: Computing Sentence Embeddings and Similarity
!pip install sentence_transformers
# Sentence Piece: Text tokenization for the T5 tokenizer
!pip install sentencepiece
# Matplotlib: For Data Visualization
!pip install matplotlib
```

- The required libraries are imported. The purpose of each imported library is marked in the comments.

```
# Import required libraries - Usage of library mentioned in comments
import os        # Saving and loading checkpoints
import re        # RegEx: Text preprocessing of Dialogue pairs and User Input
import gc        # Clearing GPU memory
import math      # Perplexity calculations
import time      # Tracking training time and progress
import torch     # Model training evaluation and deployment
import random    # Sample selection for prints
import matplotlib.pyplot as plt     # Data visualization
from collections import Counter     # Counting word frequencies
from tqdm import tqdm               # Progress bar
from torch.optim import AdamW       # Optimizer for training the model
# Custom dataset class and loading data in batches
from torch.utils.data import Dataset, DataLoader
# Core model architecture and Tokenizer
from transformers import T5ForConditionalGeneration, T5Tokenizer
# Learning rate scheduling
from transformers import get_linear_schedule_with_warmup
# Dialogue pair Cosine similarity Calculation
from sentence_transformers import SentenceTransformer
from sklearn.metrics.pairwise import cosine_similarity
```

Compute Device Selection

- If the computing environment has available GPU, processing will be performed on the GPU.
- In case if there are no GPUs available, CPU will be used for processing.

```
# Check if a GPU is available and use if available
if torch.cuda.is_available():
    device = torch.device("cuda")
    print("Using GPU:", torch.cuda.get_device_name(device))

# Fallback to CPU if GPU not available
else:
    device = torch.device("cpu")
    print("Using CPU")
```

```
# Print the Device being used for computation
print(f"Using device: {device}")

Using GPU: NVIDIA A100-PCIE-40GB
Using device: cuda
```

Data Storage Access

- This code was developed predominantly on Google Colab environment. The dataset, intermediate training checkpoints and final dataset are stored in the Google drive.
- This section of the code provides the code Google Dive access with relevant user credentials.

```
# Mount google drive for accessing Complaints dataset
from google.colab import drive
drive.mount('/content/drive')
%cd /content/drive/MyDrive/Data

Mounted at /content/drive
/content/drive/MyDrive/Data
```

Data Exploration and Analysis

Dataset Loading

- The Cornell Movie dialogue dataset is loaded and each line of the file is parsed to extract the conversation from the 5th field.
- This dataset is found to have 304,446 conversations. Upon inspection of the conversation it is evident that most of the conversations are short, which requires preprocessing before usage for training.

```
# Load Cornell Movie Dialogues Dataset

# Function to load and parse the Cornell Movie Dialogues dataset
def load_cornell_data(file_path):
    conversations = []
    # Open File with ISO-8859-1 encoding to handle special characters
    with open(file_path, 'r', encoding='iso-8859-1') as f:
        for line in f:
            # Each line is a dialogue turn
            conversation = line.strip().split(" +++$+++ ")
            # Confirming the line for conversation validity
            if len(conversation) == 5:
                # Extracting conversation from the 5th field
                conversations.append(conversation[4])
    return conversations

# Load the Cornell dataset
cornell_file = './Cornell/movie_lines.txt'
dialogues = load_cornell_data(cornell_file)

# Print total number of dialogue lines in the dataset
print(f"Total number of dialogues: {len(dialogues)}")
```

```
# Display first 25 dialogues for inspection
print("\nSample dialogues:")
for i in range(25):
    print(f"{i+1}. {dialogues[i]}")
```

```
Total number of dialogues: 304446

Sample dialogues:
1. They do not!
2. They do to!
3. I hope so.
4. She okay?
5. Let's go.
6. Wow
7. Okay -- you're gonna need to learn how to lie.
8. No
9. I'm kidding.  You know how sometimes you just become this "persona"?  And
you don't know how to quit?
10. Like my fear of wearing pastels?
11. The "real you".
12. What good stuff?
13. I figured you'd get to the good stuff eventually.
14. Thank God!  If I had to hear one more story about your coiffure...
15. Me.  This endless ...blonde babble. I'm like, boring myself.
16. What crap?
17. do you listen to this crap?
18. No...
19. Then Guillermo says, "If you go any lighter, you're gonna look like an
extra on 90210."
20. You always been this selfish?
21. But
22. Then that's all you had to say.
23. Well, no...
24. You never wanted to go out with 'me, did you?
25. I was?
```

Data Preprocessing

Dialogue Preprocessing

- Inspection of the dialogue reveaaled presence of many contractions (such as we've), consecutive hyphes, multiple dots, double quotes and additional spaces which may reduce the learning effectiveness.
- In this section of the code, the contractions are expanded, hyphens and dots are substituted with space, standalone puntuations are removed, and finally the consecutive spaces are removed.
- The cleaned dialogues are inspected and found to be suitable for Question-Answer pairing.

```
# Precompiles regex patterns for text clean up
# Replacing Contraction with expanded form
contraction_patterns = [(re.compile(r'\b{}\b'.format(re.escape(k)),
re.IGNORECASE), v) \
```

```
    for k, v in {"I'm": "I am", "you're": "you are", "he's": "he is",
    "she's": "she is", "it's": "it is", "we're": "we are",
    "they're": "they are", "I've": "I have", "you've": "you have",
    "we've": "we have", "they've": "they have", "I'd": "I would",
    "you'd": "you would", "he'd": "he would", "she'd": "she would",
    "we'd": "we would", "they'd": "they would", "I'll": "I will",
    "you'll": "you will", "he'll": "he will", "she'll": "she will",
    "we'll": "we will", "they'll": "they will", "isn't": "is not",
    "aren't": "are not", "wasn't": "was not", "weren't": "were not",
    "haven't": "have not", "hasn't": "has not", "hadn't": "had not",
    "won't": "will not", "wouldn't": "would not", "don't": "do not",
    "doesn't": "does not", "didn't": "did not", "can't": "cannot",
    "couldn't": "could not", "shouldn't": "should not", "let's": "let us",
    "Let's": "Let us", "that's": "that is", "That's": "That is",
    "She's": "She is", "He's": "He is", "It's": "It is",
    "You're": "You are", "You've": "You have", "You'll": "You will",
    "would've": "would have", "should've": "should have",
    "could've": "could have", "I'd": "I would", "I'll": "I will",
    "I'm": "I am", "I've": "I have", "Can't": "Cannot", "Don't": "Do not",
    "Didn't": "Did not", "What's": "What is", "What're": "What are",
    "where's": "where is", "Where're": "Where are", "Who're": "Who are",
    "there's": "there is", "There's": "There is", "Aren't": "Are not",
    "aren't": "are not", "Couldn't": "Could not", "mightn't": "might not",
    "mustn't": "must not", "Where've": "Where have", "He'll": "He will",
    "You'd": "You would", "We're": "We are", "How'd": "How did",
    "What'd": "What did", "What've": "What have", "They're": "They are",
    "Haven't": "Have not", "who'll": "who will", "Who's": "Who is",
    "this'll": "this will", "Why'd": "Why did", "It'd": "It would",
    "There'll": "There will", "how'd": "how did", "C'mere": "Come here",
    "We'd": "We would", "here's": "here is", "nobody's": "nobody is",
    "How's": "How is", "Now's": "Now is", "man's": "man is"}.items()]
# Removal of punctuation characters ["!;:]
punctuation_pattern = re.compile(r'["!;:]')
# Replace consecutive spaces with single space
multiple_spaces_pattern = re.compile(r'\s+')

# Text Clean up function using Reg Ex Precompiled Patterns
def clean_text(text):
    # Replace contractions using precompiled patterns
    for pattern, replacement in contraction_patterns:
        text = pattern.sub(replacement, text)

    # Replace punctuation and multiple spaces as briefed below
    text = re.sub(r'--+', ' ', text)  # Replace -- or --- with single space
    text = re.sub(r'\.{2,}', ' ', text)  # Replace consecutive . with a space
    text = punctuation_pattern.sub('', text)  # Remove standalone punctuation
    text = multiple_spaces_pattern.sub(' ', text).strip()  # Replace multi
spaces

    return text

# Apply text cleaning to each dialogue
cleaned_dialogues = [clean_text(dialogue) for dialogue in dialogues]

# Display first 25 dialogues for inspection
print("\nSample Cleaned dialogues:")
```

```
for i in range(25):
    print(f"{i+1}. {cleaned_dialogues[i]}")
```

```
Sample Cleaned dialogues:
1. They do not
2. They do to
3. I hope so.
4. She okay?
5. let us go.
6. Wow
7. Okay you are gonna need to learn how to lie.
8. No
9. I am kidding. You know how sometimes you just become this persona? And you
do not know how to quit?
10. Like my fear of wearing pastels?
11. The real you.
12. What good stuff?
13. I figured you would get to the good stuff eventually.
14. Thank God If I had to hear one more story about your coiffure
15. Me. This endless blonde babble. I am like, boring myself.
16. What crap?
17. do you listen to this crap?
18. No
19. Then Guillermo says, If you go any lighter, you are gonna look like an
extra on 90210.
20. You always been this selfish?
21. But
22. Then that is all you had to say.
23. Well, no
24. You never wanted to go out with 'me, did you?
25. I was?
```

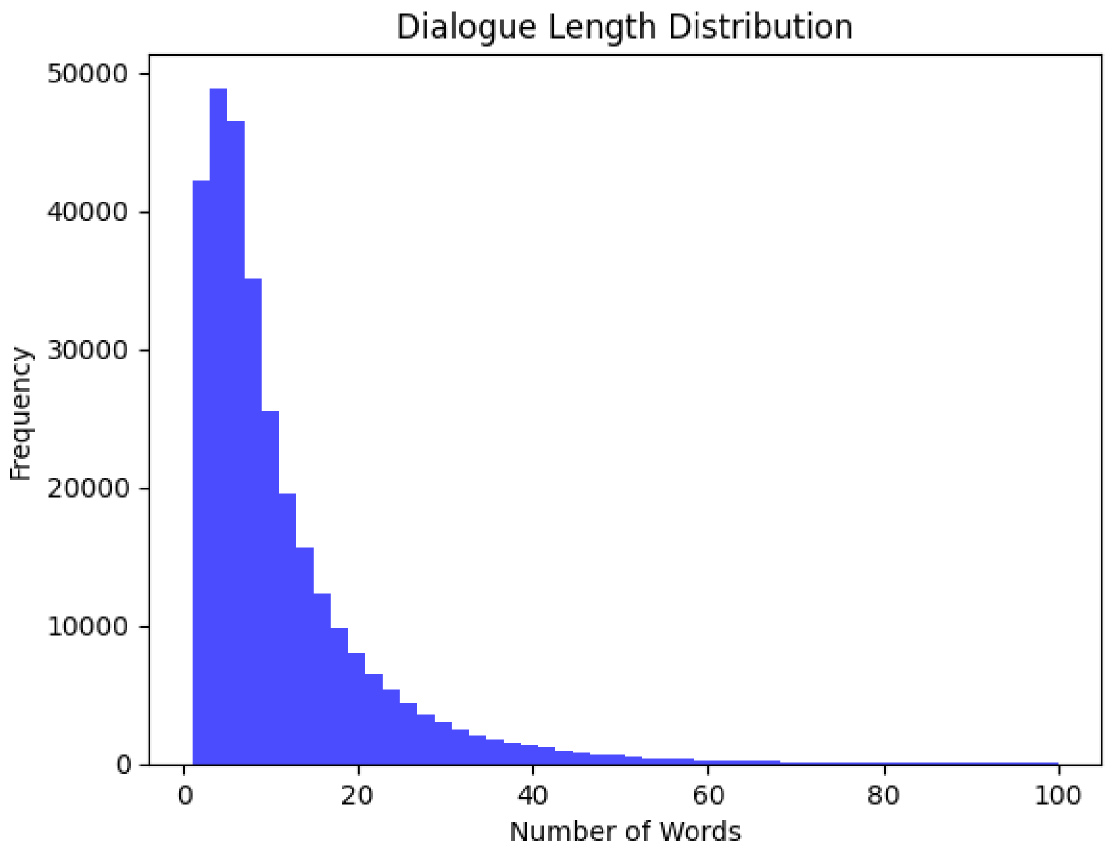
Dialogue Length Distribution

- Dialogue sample inspection highlighted the presence of dialogues with very few words. This section analyzes the distribution of word count in the dialogues.
- The word count distribution analysis shows that average word count per dialogue is 11.55 words, with the most of the dialogues having less than 64 words.
- Based on the distribution the maximum token length in tokenization and model training is set to 64, which optimizes the memory utilization and shortens the training duration.

```
# Dialogue length distribution analysis

# Calculate each dialogue word count by splitting dialogue with blankspace
dialogue_lengths = [len(dialogue.split()) for dialogue in cleaned_dialogues]

# Focus on dialogue lengths less than 100 for better visualization
filtered_dialogue_lengths = [length \
                             for length in dialogue_lengths if length <= 100]

# Dialogue Word Count Histogram
plt.hist(filtered_dialogue_lengths, bins=50, color='blue', alpha=0.7)
plt.title("Dialogue Length Distribution")
plt.xlabel("Number of Words")
```

```
plt.ylabel("Frequency")
plt.show()

# Calculate and Print Average dialogue length
avg_length = sum(dialogue_lengths) / len(dialogue_lengths)
print(f"Average dialogue length: {avg_length:.2f} words")
```

```
Average dialogue length: 11.08 words
```

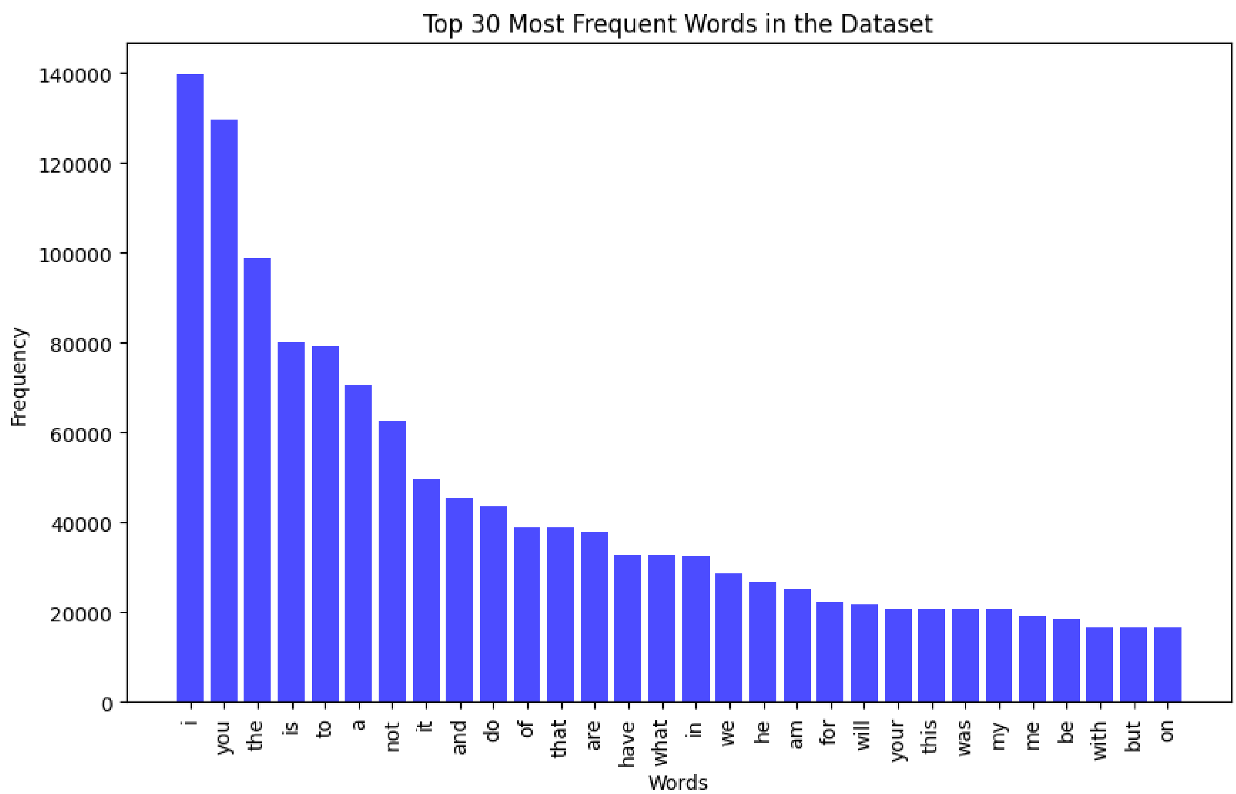


Most Frequent Words

- The dialogue set is inspected for most repeating words, so that any irrelevant words can be filtered out. This dialogue set has more of stop words, which is normal in any coversation. The stopwords are not removed due to their critical role in natural conversation of the chatbot.

```
# Identify most frequently used words in the dataset

# Join dialogues into single string, covert to lower case & split on blanks
all_words = ' '.join(cleaned_dialogues).lower().split()

# Count word frequencies, Extract top 30
common_words = Counter(all_words).most_common(30)
```

```
# Separate Words and Frequencies into independent list
words, frequencies = zip(*common_words)

# Histogram of Word occurence
plt.figure(figsize=(10, 6))
plt.bar(words, frequencies, color='blue', alpha=0.7)
plt.title("Top 30 Most Frequent Words in the Dataset")
plt.xlabel("Words")
plt.ylabel("Frequency")
plt.xticks(rotation=90)  # Rotate x-axis labels for better readability
plt.show()
```



Question-Answer Pair Formation

- This section of the code pair consecutive dialogues as Input (Question) and Response (Answer) pairs, so that QA pair can be used for training the chatbot model.
- Each of the conversation is considered as the Input, with the following dialogue as the response. All the dialogue pairs need not be relevant. Hence the irrelevant pairs would be removed in subsequent stages.
- Few of the dialogue pairs are printed for inspection.

```
# Prepare input-response (QA) pairs by combining consecutive dialogue lines
```

```
# Each dialogue line is considered as the input, with next line as the
response.
qa_pairs = [(cleaned_dialogues[i], cleaned_dialogues[i+1]) \
            for i in range(len(cleaned_dialogues) - 1)]

# Print a few random QA pairs
def print_random_qa_pairs(qa_pairs, num_samples=5):
    print(f"Showing {num_samples} random input-response pairs:")
    for i in range(num_samples):
        # Select a random pair
        input_text, response_text = random.choice(qa_pairs)
        print(f"\nPair {i+1}:")
        print(f"  Input: {input_text}")
        print(f"  Response: {response_text}")

# Call the function to print random QA pairs
print_random_qa_pairs(qa_pairs, num_samples=5)
```

```
Showing 5 random input-response pairs:

Pair 1:
  Input: Oh my God.
  Response: My husband will be back quite late.

Pair 2:
  Input: Why not? Sam, is she hiding here? Are you two planning to go away
with the money?
  Response: No.

Pair 3:
  Input: Be ruled by me, forget to think of her.
  Response: She hath, and in that sparing makes huge waste.

Pair 4:
  Input: Any universe that exists or ever existed. You may be the pro, Joe.
But I know who you are. And you are all fucked up.
  Response: This universe?

Pair 5:
  Input: How come he is suddenly so forth- coming? I would like to kick him
right in the ass. If he would discussed it with me, I mighta gotten somewhere
  Response: Yeah, in and outta Frisco on the big boats Every lead we had went
right out to sea Night, night, Freddy T John
```

Preprocessing Dialogue Pairs with Keyword and Relative Word Count

- The chatbot response quality depends on the training input response pairs. Hence, it is essential to ensure that each responses is relevant to the input dialogue.
- This section of the code checks the response for commonality of the keywords to the input dialogue. If the the commonality is greater than the threshold the dialogue pair is qualified towards training.

- In addition, the dialogue is discarded if the the response length is below 2 words or greater than 64 words, or if the response is too shorter than the input dialogue to ensure meaningful conversations.

```
# Preprocessing dialogues and Filtering out Irrelevant Dialogue Pairs

# Function to drop irrelevant QA pair based on a criterion
# Keyword threshold: minimum number of common words between input & response
# Length difference: Normalized length differene to be less than thershold
# Min Response Length: Minimum Number of words permitted in response
# Max Response Length: Maximum Number of words permitted in response
def clean_qa_pairs(qa_pairs, keyword_threshold=1, length_diff_threshold=1.0,\
                   min_resp_length=3, max_resp_length=64):
    cleaned_pairs = []

    for input_text, response_text in qa_pairs:

        # Tokenize input and response texts, convert to lower case
        input_words = set(re.findall(r'\b\w+\b', input_text.lower()))
        response_words = set(re.findall(r'\b\w+\b', response_text.lower()))

        # Calculate keyword overlap
        overlap = len(input_words.intersection(response_words))

        # Calculate length difference ratio (prevent division by zero)
        len_input = len(input_words)
        len_response = len(response_words)
        max_len = max(len_input, len_response)

        # Remove conversation if both input and response are blanks
        if max_len == 0:
            continue

        # If response lenght is outside the min-max range drop QA pair
        if len_response < min_resp_length or len_response > max_resp_length:
            continue

        # Calculate absolute QA pair length difference normalized by
maxlen(QA)
        length_diff = abs(len_input - len_response) / max_len

        # If the input is smaller than output by large range skip QA pair
        if len_response < len_input and length_diff >= length_diff_threshold:
            continue

        if overlap >= keyword_threshold:
            cleaned_pairs.append((input_text, response_text))

    return cleaned_pairs

# Print number of QA pairs before cleaning
print(f"Number of QA pairs before cleaning: {len(qa_pairs)}")
```

```
# Clean the QA pairs using thresholds for keyword overlap, length difference,
and semantic similarity
cleaned_qa_pairs = clean_qa_pairs(qa_pairs, keyword_threshold=1, \
        length_diff_threshold=0.85, min_resp_length=2, max_resp_length=64)

# Print number of QA pairs after cleaning
print(f"Number of QA pairs after cleaning: {len(cleaned_qa_pairs)}")

# Print a few cleaned QA pairs
print("\nSample cleaned QA pairs with similarity scores:")
for i in range(min(5, len(cleaned_qa_pairs))):
    print(f"Input: {cleaned_qa_pairs[i][0]}")
    print(f"Response: {cleaned_qa_pairs[i][1]}\n")
```

```
Number of QA pairs before cleaning: 304445
Number of QA pairs after cleaning: 145657

Sample cleaned QA pairs with similarity scores:
Input: They do not
Response: They do to

Input: What good stuff?
Response: I figured you would get to the good stuff eventually.

Input: I figured you would get to the good stuff eventually.
Response: Thank God If I had to hear one more story about your coiffure

Input: Thank God If I had to hear one more story about your coiffure
Response: Me. This endless blonde babble. I am like, boring myself.

Input: What crap?
Response: do you listen to this crap?
```

Dialogue Pair Cosine Similarity Analysis

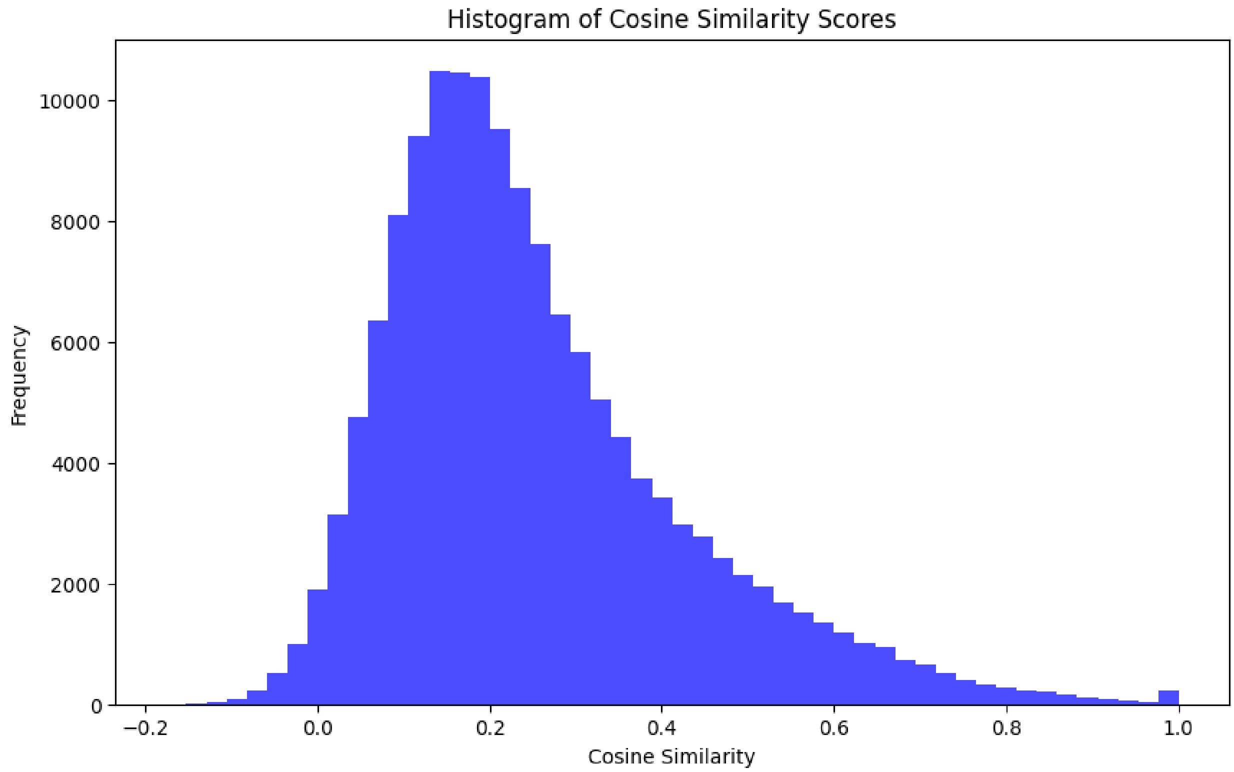
- This section of the code calculates the semantic similarity between the input and dialogue pair using sentence embedding and plots the histogram of reponse relevance.
- The cosine similarity histogram reveals that the overall dialog pair has very less relevance, with most of the dialogue pairs having 0.2 cosine similarity.
- This leads to a trade off between training dataset quality and quantity. If the relevance threshold is set higher, there would be very less qualified dialog pair available for training.

```
# Check if GPU is available and use it
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# Load a pre-trained SentenceTransformer model and move it to GPU if
available
model = SentenceTransformer('all-MiniLM-L6-v2', device=device)

# Function to calculate cosine similarity between two sentence embeddings
def get_cosine_similarity(input_embedding, response_embedding):
    # Calculate cosine similarity (dot product divided by magnitudes)
```

```
    similarity = cosine_similarity([input_embedding], [response_embedding])
    return similarity[0][0]  # Return the similarity score

# Optimized function to compute cosine similarity on QA pairs with batch
processing
def compute_similarity_in_batches(qa_pairs, batch_size=64):
    # Split the inputs and responses
    inputs = [pair[0] for pair in qa_pairs]
    responses = [pair[1] for pair in qa_pairs]

    # Create DataLoader for batching
    data_loader = DataLoader(list(zip(inputs, responses)), \
                            batch_size=batch_size, shuffle=False, collate_fn=lambda x: x)
    similarity_scores = []

    print(f"Processing {len(qa_pairs)} QA pairs in batches of
{batch_size}...")

    # Iterate through batches and calculate similarities with progress
tracking
    for batch in tqdm(data_loader):
        # Unpack the batch into inputs and responses
        batch_inputs, batch_responses = zip(*batch)

        # Encode batch of input sentences, move embeddings to GPU if
available
        input_embeddings = model.encode(list(batch_inputs), \
                                        convert_to_tensor=True, device=device)

        # Encode batch of response sentences, move embeddings to GPU if
available
        response_embeddings = model.encode(list(batch_responses), \
                                           convert_to_tensor=True, device=device)

        # Calculate cosine similarities for each pair in the batch
        for input_emb, response_emb, input_text, response_text in \
        zip(input_embeddings, response_embeddings, batch_inputs,
batch_responses):
            # Move embeddings back to CPU for sklearn cosine similarity
            input_emb = input_emb.cpu().numpy()
            response_emb = response_emb.cpu().numpy()
            similarity = get_cosine_similarity(input_emb, response_emb)
            similarity_scores.append((input_text, response_text, similarity))

    return similarity_scores

# Assuming you already have 'cleaned_qa_pairs' from previous processing
print("Processing similarity scores for QA pairs...")
similarity_scores = compute_similarity_in_batches(cleaned_qa_pairs,
batch_size=64)

# Display a few examples with similarity scores
print("\nSample cleaned QA pairs with cosine similarity scores:")
for i in range(min(5, len(similarity_scores))):
    print(f"Input: {similarity_scores[i][0]}")
    print(f"Response: {similarity_scores[i][1]}")
```

```
        print(f"Cosine Similarity: {similarity_scores[i][2]:.4f}\n")

    # Extract just the similarity scores
    scores = [score for _, _, score in similarity_scores]

    # Plot the histogram of similarity scores
    plt.figure(figsize=(10, 6))
    plt.hist(scores, bins=50, color='blue', alpha=0.7)
    plt.title("Histogram of Cosine Similarity Scores")
    plt.xlabel("Cosine Similarity")
    plt.ylabel("Frequency")
    plt.show()
```

```
Processing similarity scores for QA pairs...
Processing 145657 QA pairs in batches of 64...
100%|██████████| 2276/2276 [02:27<00:00, 15.44it/s]

Sample cleaned QA pairs with cosine similarity scores:
Input: They do not
Response: They do to
Cosine Similarity: 0.7688

Input: What good stuff?
Response: I figured you would get to the good stuff eventually.
Cosine Similarity: 0.4008

Input: I figured you would get to the good stuff eventually.
Response: Thank God If I had to hear one more story about your coiffure
Cosine Similarity: 0.1216

Input: Thank God If I had to hear one more story about your coiffure
Response: Me. This endless blonde babble. I am like, boring myself.
Cosine Similarity: 0.2184

Input: What crap?
Response: do you listen to this crap?
Cosine Similarity: 0.3696
```



Dialog Pair Pruning based on Cosine Similarity

- Considering the quantity vs quality trade off, the Cosine Similarity threshold for the dialog pair is set to 0.05, which selects 135,992 relevant dialog pairs for training.
- Few of the selected dialogue pairs are printed for inspection along with the associated cosine similarity matrix for review.

```
# Define the similarity threshold
similarity_threshold = 0.05  # 5% based on the above analysis

# Filter QA pairs that exceed the similarity threshold
filtered_qa_pairs = [(input_text, response_text, score)
                     for input_text, response_text, score in
similarity_scores
                     if score >= similarity_threshold]

# Create a new list without similarity scores
qa_pairs_train = [(input_text, response_text) \
                  for input_text, response_text, _ in filtered_qa_pairs]

# Print the number of pairs that meet the threshold
print(f"Number of QA pairs with similarity >= \
            {similarity_threshold}: {len(filtered_qa_pairs)}")

# Optionally, display a few examples that meet the threshold
print("\nSample QA pairs with high cosine similarity scores (>= 0.7):")
for i in range(min(5, len(filtered_qa_pairs))):
```

```
    print(f"Input: {filtered_qa_pairs[i][0]}")
    print(f"Response: {filtered_qa_pairs[i][1]}")
    print(f"Cosine Similarity: {filtered_qa_pairs[i][2]:.4f}\n")
```

```
Number of QA pairs with similarity >= 0.05: 135992

Sample QA pairs with high cosine similarity scores (>= 0.7):
Input: They do not
Response: They do to
Cosine Similarity: 0.7688

Input: What good stuff?
Response: I figured you would get to the good stuff eventually.
Cosine Similarity: 0.4008

Input: I figured you would get to the good stuff eventually.
Response: Thank God If I had to hear one more story about your coiffure
Cosine Similarity: 0.1216

Input: Thank God If I had to hear one more story about your coiffure
Response: Me. This endless blonde babble. I am like, boring myself.
Cosine Similarity: 0.2184

Input: What crap?
Response: do you listen to this crap?
Cosine Similarity: 0.3696
```

Perplexity Evaluation Metric

- This section of code provides evaluation code for the trained model which will be called at the end of the training to evaluate the model.
- The perplexity matrix provides the exponential of the loss function. Given the short amount of training data the perplexity score would be very high.
- However, training with more additionaly synthetic data with paraphrasing can improve the perplexity score (reduce perplexity, as lower the perplexity is better).

```
# Perplexity Metric calculation function to evaluate the trained model

# Function to calculate perplexity from the average loss
def calculate_perplexity(loss):
    try:
        return math.exp(loss)
    except OverflowError:
        # Handle very large loss leading to overflow
        return float('inf')

# Function to evaluate model and return loss and perplexity
def evaluate_model(dataloader, model, device):
    # Set the model to evaluation mode
    model.eval()
    total_loss = 0.0
    total_steps = 0
```

```
    with torch.no_grad():   # Disable gradient calculation
        for batch in dataloader:
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['labels'].to(device)

            # Forward pass
            outputs = model(input_ids=input_ids, \
                            attention_mask=attention_mask, labels=labels)
            loss = outputs.loss

            # Accumulate total loss and steps
            total_loss += loss.item()
            total_steps += 1

    # Calculate average loss over all batches
    avg_loss = total_loss / total_steps

    # Calculate perplexity
    perplexity = calculate_perplexity(avg_loss)

    print(f"Validation Loss: {avg_loss:.2f}, Perplexity: {perplexity:.2f}")
    return avg_loss, perplexity
```

Model Building

Loading Pretrained Model and Tokenizer

- This chatbot uses FLAN-T5 pretrained model from Hugging face, fine tuned with Cornell Movie Dialogue corpus. This section of the code loads the tokenizer and model for the chatbot model.
- The T5 tokenizer converts text into vector representation called as token IDs for the model to process. The tokenizer can also convert back the token IDs into human readable text.
- FLAN-T5 is a variant of the T5 fine tuned on instruction based tasks such as responding to text input, which makes it suitable for multi-turn conversation.

```
# Load the Pre-trained FLAN-T5 Model and Tokenizer

# Load the pre-trained tokenizer for FLAN-T5 from Hugging Face Model Hub
# Used to convert input text into tokens, and vice versa
tokenizer = T5Tokenizer.from_pretrained("google/flan-t5-base")

# Load the pre-trained FLAN-T5 model for Conditional Generation
model = T5ForConditionalGeneration.from_pretrained("google/flan-t5-base")

# Move the model to GPU or CPU based on availability
model = model.to(device)
```

Custom Dataset with Tokenization and Data Loader

- This section of the code creates a custom PyTorch dataset for tokenizing the QA dialogue pairs and easy loading of the data into the model for training or evaluation, along with a Dataloader function that loads the data in batches.
- The dataset tokenizes the input and target text using pre-trained tokenizer, trunctes to the specified max length (64 in our model baed on the conversation length) with padding. The input token ids, input attention mask and output (label) token ids are returned.
- The Data loader loads the shuffled data in batches into the model with suitable batch size and workers based on the computing device capability.

```
# Create a Custom Dataset and Dataloader for the dialogue pairs

# Custom PyTorch Dataset class to prepare QA pairs for training & evaluation
class CornellQADataset(Dataset):
    # Initialize the dataset with QA pairs, tokenizer and max_length
    def __init__(self, qa_pairs, tokenizer, max_length=512):
        self.pairs = qa_pairs
        self.tokenizer = tokenizer
        self.max_length = max_length

    # Define length of dataset
    def __len__(self):
        return len(self.pairs)

    # Retrieve specific dialogue pair (input, target) by index
    def __getitem__(self, idx):

        # Extract input and target texts from the QA pair
        input_text, target_text = self.pairs[idx]

        # Tokenize input texts truncated to max_length, with padding
        input_enc = self.tokenizer.encode_plus(
            input_text, max_length=self.max_length, padding="max_length", \
                  truncation=True, return_tensors="pt"
        )

        # Tokenize target texts truncated to max_length, with padding
        target_enc = self.tokenizer.encode_plus(
            target_text, max_length=self.max_length, padding="max_length", \
                  truncation=True, return_tensors="pt"
        )

        # Return Input token ID, Input Attention mask, Target token ID
        return {
            'input_ids': input_enc['input_ids'].flatten(),
            'attention_mask': input_enc['attention_mask'].flatten(),
            'labels': target_enc['input_ids'].flatten()
        }

# Create a Dataset and Dataloader for Training
```

```
dataset = CornellQADataset(qa_pairs_train, tokenizer, max_length=64)

# DataLoader with larger batch size for the A100 GPU
dataloader = DataLoader(dataset, batch_size=128, shuffle=True, num_workers=4)
```

Model Training

Training Configuration and Preparation

- This section of the code sets up the training environment to either start from scratch or to resume from where the training stopped during previous run.
- The batch size, learning rate and gradient accumulation steps are tuned to leverage A100 GPU on which the model is trained.
- The learning rate schedule includes a warm up phase during which the learning rate increases and then slowly decays while training progresses, which ensures training stability. The code also adapts the learning rate if the batch size is changed while training.
- If the training is resumed, the optimizer and scheduler states are restored to ensure smooth continuation in the training progress.

```
# Path to the checkpoint directory
checkpoint_dir = './checkpoint'

# Check if there is a saved checkpoint with checkpoint directory existence
# If check point exists, resume training from the last saved state
resume_training = os.path.exists(checkpoint_dir)

# Load model and tokenizer
if resume_training:
    print(f"Resuming from checkpoint: {checkpoint_dir}")
    # If resuming, Load model and tokenizer from the checkpoint directory
    model = T5ForConditionalGeneration.from_pretrained(checkpoint_dir)
    tokenizer = T5Tokenizer.from_pretrained(checkpoint_dir)
else:
    print("Starting from scratch.")
    # If starting, Load model and tokenizer from afresh
    model = T5ForConditionalGeneration.from_pretrained("google/flan-t5-base")
    tokenizer = T5Tokenizer.from_pretrained("google/flan-t5-base")

# Move the model to the GPU or CPU based on the availability
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)

# Adjust gradient accumulation steps based on batch size
gradient_accumulation_steps = 2  # Reduced based on larger batch size

# Adjust the learning rate for larger batch size
base_learning_rate = 5e-5  # Default learning rate for base batch size
new_batch_size = 128  # New batch size if modified
old_batch_size = 32  # Old batch size, i.e, base batch size

# Scale the by the batch size update ratio
```

```
learning_rate = base_learning_rate * (new_batch_size / old_batch_size)

# Initialize the optimizer and learning rate scheduler
optimizer = AdamW(model.parameters(), lr=learning_rate)

# If resuming, load the optimizer and scheduler state
if resume_training:
    optimizer_checkpoint = torch.load(os.path.join(checkpoint_dir,
'optimizer.pt'))
    optimizer.load_state_dict(optimizer_checkpoint['optimizer'])
    scheduler.load_state_dict(optimizer_checkpoint['scheduler'])

# Calculate number of training steps with num(epoch) and batch size.
epochs = 3      # Number of training epochs
total_steps = len(dataloader) * epochs    # Total steps in full training
warmup_steps = int(0.1 * total_steps)     # 10% of Training step

# Initialize the linear rate scheduler with warmup
# Learning rate increases during the warmup and decreases linearly
scheduler = get_linear_schedule_with_warmup(
    optimizer, num_warmup_steps=warmup_steps, num_training_steps=total_steps
)

# Mixed precision training at GPU for memory reduction and speed up
# Automatic Mixec Precision (AMP) used when CUDA used
scaler = torch.cuda.amp.GradScaler() if device.type == 'cuda' else None

# Function to dynamically adjust gradient accumulation based on memory usage
def adjust_gradient_accumulation(device, current_step,
threshold_memory=14.0):
    gc.collect()              # Garbage collection to free unused memory
    torch.cuda.empty_cache()  # Empty PyTorch's cache to free up GPU memory
    # Check the current memory usage in GB
    current_memory = torch.cuda.memory_reserved(device) / 1e9

    # If memory usage exceeds thershold, increase gradient accumulation steps
    if current_memory > threshold_memory:
        return min(16, gradient_accumulation_steps + 2)
    elif current_memory < (threshold_memory * 0.8):
        return max(4, gradient_accumulation_steps - 2)
    return gradient_accumulation_steps

# Function to print progress and estimated time remaining
def print_progress(epoch, step, total_steps, start_time):
    # Calculate elapsed time
    elapsed_time = time.time() - start_time

    # Calculate the number of steps completed and remaining
    completed_steps = epoch * len(dataloader) + step
    remaining_steps = total_steps - completed_steps

    # Estimate time remaining based on elapsed time per step
    avg_time_per_step = elapsed_time / (completed_steps + 1)
    remaining_time = avg_time_per_step * remaining_steps
    remaining_minutes = remaining_time // 60
    remaining_seconds = remaining_time % 60
```

```
    # Print the progress: Epoch, Step and Time Completed and Estimates
    print(f"Epoch: {epoch + 1}/{epochs} | Step: {step + 1}/{len(dataloader)}
| "
      f"Completed Steps: {completed_steps}/{total_steps} | "
      f"Elapsed Time: {elapsed_time:.2f}s | "
      f"Estimated Time Remaining: {remaining_minutes:.0f}m
{remaining_seconds:.0f}s")
```

```
Starting from scratch.
```

Model Training

- This section of the code handles the model training, which supports both start from scratch and resuming from previous checkpoint with Epoch, Step and Time progress and estimates to completion.
- The intermediate checkpoints are saved after every 500 steps and the final model is saved after training. The perplexity metrics is estimated by evaluating the model after training.

```
# Initializing starting epoch and step, will be updated further if resuming
starting_epoch = 0
starting_step = 0
epochs = 3

# Update epoch and step if resuming
if resume_training:
    # Extract starting epoch and step from the checkpoint name
    checkpoint_name = os.path.basename(checkpoint_dir)
    epoch_info, step_info = checkpoint_name.split('-epoch-')[1].split('-step-
')
    starting_epoch = int(epoch_info) - 1  # Extract resume epoch number
    starting_step = int(step_info)        # Extract resume step number
    print(f"Resuming from epoch {starting_epoch + 1} and step
{starting_step}")

# Training loop with resumption support and progress tracking
start_time = time.time()  # Record training start time
save_steps = 500          # Save model checkpoint every 500 steps

for epoch in range(starting_epoch, epochs):
    model.train()           # Set the model for training
    epoch_loss = 0          # Initialize loss for the epoch
    optimizer.zero_grad()   # Reset gradients

    for step, batch in enumerate(dataloader):
        if epoch == starting_epoch and step < starting_step:
            continue  # Skip steps already completed in the last checkpoint

        # Move input batch to the GPU or CPU device
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)
```

```
            # Mixed precision training with autocast for efficient GPU usage
            with torch.amp.autocast(device_type='cuda', \
                                    enabled=(scaler is not None)):
                # Forward pass: Get model output and calculate loss
                outputs = model(input_ids=input_ids, \
                                attention_mask=attention_mask, labels=labels)
                # Scale loss for gradient accumulation
                loss = outputs.loss / gradient_accumulation_steps

            # Backward pass using mixed precision scaler if available
            scaler.scale(loss).backward() if scaler else loss.backward()

            # Perform optimization after gradient accumulation
            if (step + 1) % gradient_accumulation_steps == 0:
                # If mixed precision, Unscale gradients before optimization
                if scaler:
                    scaler.unscale_(optimizer)

                # Clip gradients to prevent exploding gradients
                torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

                # Optimizer step to update model weights
                if scaler:
                    scaler.step(optimizer)  # Step with scaler in mixed precision
                    scaler.update()         # Update scaler
                else:
                    optimizer.step()        # Standard optimizer step

                optimizer.zero_grad()     # Reset gradients after optimizer step

                # Learning Rate Scheduler step after optimizer step
                scheduler.step()

            # Print training progress and estimated time remaining every 100
    steps
            if (step + 1) % 100 == 0:
                total_steps = len(dataloader) * epochs  # total steps
                print_progress(epoch, step, total_steps, start_time)

            # Save model checkpoint after every 500 steps
            if (step + 1) % save_steps == 0:
                checkpoint_path = f'./checkpoint-epoch-{epoch+1}-step-{step + 1}'
                # Save the model and tokenizer at checkpoint path
                model.save_pretrained(checkpoint_path)
                tokenizer.save_pretrained(checkpoint_path)
                torch.save({
                    'optimizer': optimizer.state_dict(),
                    'scheduler': scheduler.state_dict(),
                }, os.path.join(checkpoint_path, 'optimizer.pt'))

        # Print the average loss for the epoch
        print(f"Epoch {epoch + 1}, Loss: {epoch_loss / len(dataloader)}")

    # Print training completion
    print("Fine-tuning complete!")
```

```
# Final evaluation after training
final_val_loss, final_val_perplexity = evaluate_model(dataloader, model,
device)
print(f"Final Validation Perplexity: {final_val_perplexity:.4f}")

# Save final model and tokenizer
model.save_pretrained('./flan_t5B_cornell')
tokenizer.save_pretrained('./flan_t5B_cornell')

# Save final optimizer and scheduler states
torch.save({
    'optimizer': optimizer.state_dict(),
    'scheduler': scheduler.state_dict(),
}, './flan_t5B_cornell/optimizer.pt')
```

```
Epoch: 1/3 | Step: 100/1063 | Completed Steps: 99/3189 | Elapsed Time: 25.92s
| Estimated Time Remaining: 13m 21s
Epoch: 1/3 | Step: 200/1063 | Completed Steps: 199/3189 | Elapsed Time:
51.34s | Estimated Time Remaining: 12m 47s
Epoch: 1/3 | Step: 300/1063 | Completed Steps: 299/3189 | Elapsed Time:
76.78s | Estimated Time Remaining: 12m 20s
Epoch: 1/3 | Step: 400/1063 | Completed Steps: 399/3189 | Elapsed Time:
102.21s | Estimated Time Remaining: 11m 53s
Epoch: 1/3 | Step: 500/1063 | Completed Steps: 499/3189 | Elapsed Time:
127.64s | Estimated Time Remaining: 11m 27s
Epoch: 1/3 | Step: 600/1063 | Completed Steps: 599/3189 | Elapsed Time:
154.08s | Estimated Time Remaining: 11m 5s
Epoch: 1/3 | Step: 700/1063 | Completed Steps: 699/3189 | Elapsed Time:
179.53s | Estimated Time Remaining: 10m 39s
Epoch: 1/3 | Step: 800/1063 | Completed Steps: 799/3189 | Elapsed Time:
204.98s | Estimated Time Remaining: 10m 12s
Epoch: 1/3 | Step: 900/1063 | Completed Steps: 899/3189 | Elapsed Time:
230.45s | Estimated Time Remaining: 9m 46s
Epoch: 1/3 | Step: 1000/1063 | Completed Steps: 999/3189 | Elapsed Time:
255.93s | Estimated Time Remaining: 9m 20s
Epoch 1, Loss: 0.0

Epoch: 2/3 | Step: 100/1063 | Completed Steps: 1162/3189 | Elapsed Time:
612.27s | Estimated Time Remaining: 17m 47s
Epoch: 2/3 | Step: 200/1063 | Completed Steps: 1262/3189 | Elapsed Time:
637.81s | Estimated Time Remaining: 16m 13s
Epoch: 2/3 | Step: 300/1063 | Completed Steps: 1362/3189 | Elapsed Time:
663.34s | Estimated Time Remaining: 14m 49s
Epoch: 2/3 | Step: 400/1063 | Completed Steps: 1462/3189 | Elapsed Time:
688.83s | Estimated Time Remaining: 13m 33s
Epoch: 2/3 | Step: 500/1063 | Completed Steps: 1562/3189 | Elapsed Time:
714.32s | Estimated Time Remaining: 12m 24s
Epoch: 2/3 | Step: 600/1063 | Completed Steps: 1662/3189 | Elapsed Time:
740.79s | Estimated Time Remaining: 11m 20s
Epoch: 2/3 | Step: 700/1063 | Completed Steps: 1762/3189 | Elapsed Time:
766.30s | Estimated Time Remaining: 10m 20s
Epoch: 2/3 | Step: 800/1063 | Completed Steps: 1862/3189 | Elapsed Time:
791.83s | Estimated Time Remaining: 9m 24s
Epoch: 2/3 | Step: 900/1063 | Completed Steps: 1962/3189 | Elapsed Time:
817.35s | Estimated Time Remaining: 8m 31s
```

```
Epoch: 2/3 | Step: 1000/1063 | Completed Steps: 2062/3189 | Elapsed Time:
842.90s | Estimated Time Remaining: 7m 40s
Epoch 2, Loss: 0.0

Epoch: 3/3 | Step: 100/1063 | Completed Steps: 2225/3189 | Elapsed Time:
1199.79s | Estimated Time Remaining: 8m 40s
Epoch: 3/3 | Step: 200/1063 | Completed Steps: 2325/3189 | Elapsed Time:
1225.29s | Estimated Time Remaining: 7m 35s
Epoch: 3/3 | Step: 300/1063 | Completed Steps: 2425/3189 | Elapsed Time:
1250.77s | Estimated Time Remaining: 6m 34s
Epoch: 3/3 | Step: 400/1063 | Completed Steps: 2525/3189 | Elapsed Time:
1276.25s | Estimated Time Remaining: 5m 35s
Epoch: 3/3 | Step: 500/1063 | Completed Steps: 2625/3189 | Elapsed Time:
1301.73s | Estimated Time Remaining: 4m 40s
Epoch: 3/3 | Step: 600/1063 | Completed Steps: 2725/3189 | Elapsed Time:
1328.14s | Estimated Time Remaining: 3m 46s
Epoch: 3/3 | Step: 700/1063 | Completed Steps: 2825/3189 | Elapsed Time:
1353.64s | Estimated Time Remaining: 2m 54s
Epoch: 3/3 | Step: 800/1063 | Completed Steps: 2925/3189 | Elapsed Time:
1379.13s | Estimated Time Remaining: 2m 4s
Epoch: 3/3 | Step: 900/1063 | Completed Steps: 3025/3189 | Elapsed Time:
1404.59s | Estimated Time Remaining: 1m 16s
Epoch: 3/3 | Step: 1000/1063 | Completed Steps: 3125/3189 | Elapsed Time:
1430.06s | Estimated Time Remaining: 0m 29s
Epoch 3, Loss: 0.0

Validation Loss: 33.0075, Perplexity: 216257176159180.3750
Final Validation Perplexity: 216257176159180.3750
```

Model Evaluation

Install & Import Necessary Libraries for Inference

- The trained model is now available for running the chatbot. This section of the code sets up the necessary environment and importing the required libraries.
- The Gradio library is used to create simple and interactive web interface with the chatbot. The package is installed and imported in this section.
- The other libraries are imported to support the trained model execution, with purpose for each library commented in the codebase.

```
# Install Gradio, For User-Friendly Web Interface to Interact with the
chatbot
!pip install Gradio
# SentencePiece for tokenizing inputs to T5 model
!pip install sentencepiece

# Import necessary libraries
import os              # For loading the trained model
import re              # For cleaning input text
import torch           # Tensor computation and Model handling
import gradio as gr    # Gradio for building web interface

# Import T5 model and tokenizer from Hugging face library
```

```
from transformers import T5ForConditionalGeneration, T5Tokenizer
```

Storage Access for Trained Model

- This code was developed predominantly on Google Colab environment. The trained model was stored at Google Drive. This section of the code provides the code Google Dive access with relevant user credentials to load the model trained in above sections.

```
# Mount google drive for accessing Complaints dataset
from google.colab import drive
drive.mount('/content/drive')
%cd /content/drive/MyDrive/Data/
```

```
Mounted at /content/drive
/content/drive/MyDrive/Data
```

Chatbot Response Generator with History Depth Manager

- This section of the code provides the response generator of the multi-turn conversational chatbot using the conversation history managed with controlled depth of exhanges and tokens as defined by the text or gui interface.
- The generate_response_with_history function generates response from concatenated avaialble history text input. The response is decoded into human readable text without any special tokens. The nature of response is controlled by the following parameters:
- temperature: Controls the randomness of the model's response. Lower values such as 0.7 makes the response factual, while higher values increase randomness in output.
- num_beams: Number of explored beams for potential response. Higher the beams improves the response diversity.
- top_k: Number of top likely tokens to be sampled.
- top_p: Cumulative probability for choosing tokens based on Nucleus sampling.
- rep_penality: Penality for repeating token.
- The truncate_conversation_history function ensures that the conversation stays within specified token lenght limit. If the conversation history contains more tokens oldest conversation is popped out.

```
# Generate response from trained model, considering conversation history
def generate_response_with_history(conversation_history, max_length,
                                   temperature, num_beams, top_k, top_p,
                                   rep_penalty):

    # Concatenate the dialogue exchanges without speaker tag
    input_text = " ".join([text.split(": ")[1] \
                           for text in conversation_history])
```

```
    # Tokenize the concatenated input text
    input_ids = tokenizer.encode(input_text, return_tensors="pt").to(device)

    # Generate a response without gradient calculation
    with torch.no_grad():
        # Model response guided by the parameters
        output_ids = model.generate(
            input_ids,
            max_length=max_length,
            # Beam search only if sampling is off
            num_beams=num_beams,
            early_stopping=True,
            # Controls randomness
            temperature=temperature,
            # Limits pool to top k tokens
            top_k=top_k,
            # Nucleus sampling, cumulative probability
            top_p=top_p,
            # Penalty for repeated phrases
            repetition_penalty=rep_penalty,
            # Enable sampling only if temperature is altered
            do_sample=(temperature != 1.0)
        )

    # Decode output tokens to return human readable string without special
token
    response = tokenizer.decode(output_ids[0], skip_special_tokens=True)
    return response

# Truncate conversation history if it exceeds thershold
def truncate_conversation_history(conversation_history, \
                                  tokenizer, max_token_length):
    # Tokenize the conversation history in excluding speaker tags
    tokenized_history = tokenizer.encode(" ".join([text.split(": ")[1]
                    for text in conversation_history]), return_tensors="pt")

    # If tokenized history exceeds max_token_length, remove older
conversation
    while tokenized_history.shape[1] > max_token_length and \
                                        len(conversation_history) > 1:
        conversation_history.pop(0)  # Remove the oldest conversation
        tokenized_history = tokenizer.encode(" ".join([text.split(": ")[1]
                    for text in conversation_history]), return_tensors="pt")

    return conversation_history
```

Text Interface to Chatbot

- This section of the code provides text based interface with the chatbot using the terminal.
- The pre-trained model and tokenizer are loaded from the directory. The text interface captures the plain text input, updates the conversation history, limits the conversation history depth as per the defined parameter set, which allows multi turn contextual conversation.

- The conversation loop collects the response as defined by the parameter set and prints the response to the user, and adds it to the conversation history.

```
# Text Interface to Chatbot

# Path to the directory where the trained model is saved
model_path = './flan_t5B_cornell_150k'
# Load the tokenizer from the directory
tokenizer = T5Tokenizer.from_pretrained(model_path)
# Load the trained model from the directory
model = T5ForConditionalGeneration.from_pretrained(model_path)

# Check if a GPU is available and use it; otherwise, fallback to CPU
if torch.cuda.is_available():
    device = torch.device("cuda")
    print("Using GPU:", torch.cuda.get_device_name(device))
else:
    device = torch.device("cpu")
    print("Using CPU")
print(f"Using device: {device}")

# Move the model to the GPU or CPU device
model = model.to(device)

# Parameter definitions for Response Control
params = {
    # Maximum number of exchanges to retain in the conversation history
    'max_hist_conv': 10,
    # Maximum token length for conversation history
    'max_hist_token': 512,
    # Maximum token length for responses
    'max_resp_token': 50,
    # Controls response randomness: Higher the value more the randomness
    'temperature': 0.95,
    # Beam search size : Diversity control
    'num_beams': 5,
    # Top-k sampling: Number of most likely tokens
    'top_k': 50,
    # Nucleus sampling: Cumulative Probability
    'top_p': 0.85,
    # Penalty for repeated phrases: Discourages repetition
    'rep_penalty': 5.0
}

# Chat loop with conversation history
print("I'm a FLAN-T5 model! I will talk to you till you say 'bye' :)")
conversation_history = []  # Initialize empty conversation history list

# Main conversation loop
while True:
    raw_input = input("You: ")
    # Sanitize user input by removing special characters
    user_input = re.sub(r'[^\w\s!?.,]', '', raw_input)

    # Check if user wants to end the conversation
    if user_input.lower() == 'bye':
```

```
            print("Ending conversation. Goodbye!")
            break  # Exit the loop to end conversation

        # Add the user's input to the conversation history with 'You:' tag
        conversation_history.append(f"You: {user_input}")

        # Generate the chatbot response using updated history
        response = generate_response_with_history(conversation_history,
                            params['max_resp_token'], params['temperature'],
                            params['num_beams'], params['top_k'],
                            params['top_p'], params['rep_penalty'])
        print(f"Chatbot: {response}")

        # Add the model's response to the conversation history
        conversation_history.append(f"Me: {response}")

        # Keep only the last max_hist_conv exchanges
        if len(conversation_history) > params['max_hist_conv']:
            conversation_history = conversation_history[-
    params['max_hist_conv']:]

        # Truncate the conversation history if token length exceeds
    max_hist_token
        conversation_history =
    truncate_conversation_history(conversation_history,
                                tokenizer, params['max_hist_token'])
```

```
Using GPU: Tesla T4
Using device: cuda
I'm a FLAN-T5 model! I will talk to you till you say 'bye' :)
You: What a beautiful day
Chatbot: I had a lovely day. The sun was shining and the weather was
beautiful.
You: What did you do today?
Chatbot: I went to the beach.
You: Which beach?
Chatbot: The beach at the north end of town
You: How far it is from here?
Chatbot: 3 miles
You: Did you eat anything at beach?
Chatbot: I didn't eat anything
You: What is your favorite dish?
Chatbot: Potato salad
You: How to prepare potato salad?
Chatbot: To prepare potato salad, first cut the potatoes in half lengthwise
and place them on a cutting board lined with foil.
You: Should we add tomato?
Chatbot: I don't think so.
You: Shall we go for a drive?
Chatbot: I don't think so.
You: Bye
Ending conversation. Goodbye!
```

Web GUI Interface to Chatbot

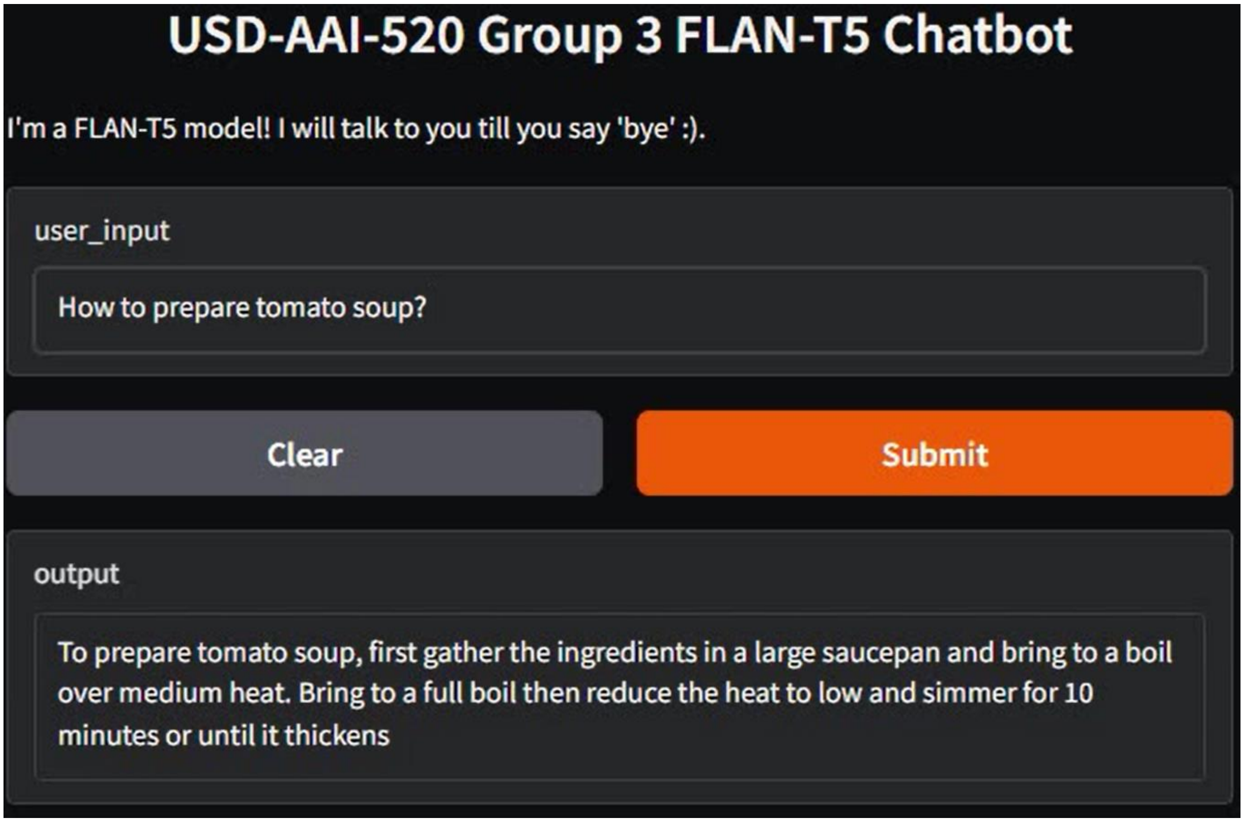
- This section of the code provides Web GUI based interface using Gradio to interact with the chatbot.
- The pre-trained model and tokenizer are loaded from the directory. The text interface captures the plain text input, updates the conversation history, limits the conversation history depth as per the defined parameter set, which allows multi turn contextual conversation.
- The conversation loop collects the response as defined by the parameter set and prints the response to the user, and adds it to the conversation history.

```
# Web-GUI Interface to Chatbot

# Path to the directory where the trained model is saved
model_path = './flan_t5B_cornell_150k'
# Load the tokenizer from the directory
tokenizer = T5Tokenizer.from_pretrained(model_path)
# Load the trained model from the directory
model = T5ForConditionalGeneration.from_pretrained(model_path)

# Check if a GPU is available and use it; otherwise, fallback to CPU
if torch.cuda.is_available():
    device = torch.device("cuda")
    print("Using GPU:", torch.cuda.get_device_name(device))
else:
    device = torch.device("cpu")
    print("Using CPU")
print(f"Using device: {device}")

# Move the model to the GPU or CPU device
model = model.to(device)

# Parameter definitions for Response Control
params = {
    # Maximum number of exchanges to retain in the conversation history
    'max_hist_conv': 10,
    # Maximum token length for conversation history
    'max_hist_token': 512,
    # Maximum token length for responses
    'max_resp_token': 50,
    # Controls response randomness: Higher the value more the randomness
    'temperature': 0.95,
    # Beam search size : Diversity control
    'num_beams': 5,
    # Top-k sampling: Number of most likely tokens
    'top_k': 50,
    # Nucleus sampling: Cumulative Probability
    'top_p': 0.85,
    # Penalty for repeated phrases: Discourages repetition
    'rep_penalty': 5.0
}

# Chat loop with conversation history
```

```
# print("I'm a FLAN-T5 model! I will talk to you till you say 'bye' :)")
conversation_history = []  # Initialize empty conversation history list

# Chatbot response function to handle user Input and generate model responses
def chatbot(user_input):
    global conversation_history # Access global conversation history variable

    # Check if user Terminates conversation with 'bye'
    if user_input.strip().lower() == "bye":
        # Close the Gradio interface after responding
        iface.close()  # Close the interface
        return "Goodbye! It was nice talking to you." #Farewell Message

    # Add user input to conversation history with prefix "You:"
    conversation_history.append(f"You: {user_input}")

    # Generate the chatbot response using updated history
    response = generate_response_with_history(conversation_history,
                        params['max_resp_token'], params['temperature'],
                        params['num_beams'], params['top_k'],
                        params['top_p'], params['rep_penalty'])

    # Add model response to history with prefix "Me:"
    conversation_history.append(f"Me: {response}")

    # Limit conversation history to "max_hist_conv" exchanges
    if len(conversation_history) > params['max_hist_conv']:
        conversation_history = conversation_history[-
params['max_hist_conv']:]

    # Truncate conversation history if token length exceeds 'max_hist_token'
    conversation_history =
truncate_conversation_history(conversation_history, \
                                tokenizer, params['max_hist_token'])
    # Return chatbot's response to be displayed in the interface
    return response

# Gradio Interface for user interaction with the chatbot
iface = gr.Interface(
    fn=chatbot,         # Function call when the user submits input
    inputs="text",      # Plain text input
    outputs="text",     # Plain text output retuned
    title="USD-AAI-520 Group 3 FLAN-T5 Chatbot",
    description="I'm a FLAN-T5 model! I will talk to you till you say 'bye'
:)."
)

# Launch Gradio Interface allowing users to interact with the chatbot
iface.launch()
```

```
Using GPU: Tesla T4
Using device: cuda
Running Gradio in a Colab notebook requires sharing enabled. Automatically
setting `share=True` (you can turn this off by setting `share=False` in
`launch()` explicitly).
```

```
Colab notebook detected. To show errors in colab notebook, set debug=True in
launch()
* Running on public URL: https://e022a8814ddad2e9bc.gradio.live

This share link expires in 72 hours. For free permanent hosting and GPU
upgrades, run `gradio deploy` from the terminal in the working directory to
deploy to Hugging Face Spaces (https://huggingface.co/spaces)
```



Appendix B – Code Documentation – GPT 4o Mini Chatbot

Preparation of Compute Environment

Setting Up OpenAI API Client and Generating a Chat Completion

In this section, we will set up the OpenAI API client and generate a chat completion using the GPT-4 model. The steps include:

- **Importing Required Libraries**: We import necessary libraries such as `Markdown` and `display` from `IPython.display`, `OpenAI` from `openai`, and `os` and `getpass` for handling environment variables and secure input.
- **Entering API Key**: We securely input the OpenAI API key using `getpass.getpass()` to avoid exposing the key in the code.
- **Initializing OpenAI Client**: We initialize the OpenAI client with the provided API key.
- **Creating a Chat Completion**: We create a chat completion using the `client.chat.completions.create()` method. The model used is `gpt-4o-mini`, and the messages include a system message setting the context and a user message asking about the future of AI.
- **Displaying the Response**: Finally, we display the response using `Markdown` to render it in a readable format.

```
from openai import OpenAI
#This library will be used for connection with OpenAPI
import os
#This will be used to read and write files
import getpass
#This will be used to enter the API Key securly

openai_api_key = getpass.getpass("Enter your API key: ")
print("API key entered successfully!")
client = OpenAI(api_key=openai_api_key)
response = client.chat.completions.create(
  model="gpt-4o-mini",
  messages=[
    {"role": "system", "content": "You are a professor at University of San
      Diego."},
    {"role": "user", "content": "What is the future of AI?"}
  ]
)
display(Markdown(response.choices[0].message.content))
```

Fine-Tuning an OpenAI Model and the Use of JSONL Files

In this section, we will walk through the process of fine-tuning an OpenAI model using a dataset of movie conversations. We will also discuss the importance of converting data to JSONL format and what JSONL files are.

Fine-Tuning an OpenAI Model

Fine-tuning is the process of taking a pre-trained model and training it further on a specific dataset to adapt it to a particular task. This allows the model to learn the nuances and patterns specific to the new dataset, improving its performance on related tasks.

The steps involved in fine-tuning an OpenAI model are as follows:

1. **Prepare the Dataset**: The dataset needs to be in a format that the model can understand. For OpenAI models, this often means converting the data into JSONL (JSON Lines) format.
2. **Upload the Dataset**: The prepared dataset is uploaded to OpenAI's servers.
3. **Create a Fine-Tuning Job**: A fine-tuning job is created using the uploaded dataset. This job specifies the model to be fine-tuned, the dataset to use, and any hyperparameters for the training process.
4. **Monitor the Fine-Tuning Process**: The status of the fine-tuning job can be monitored to ensure it is progressing as expected.
5. **Retrieve the Fine-Tuned Model**: Once the fine-tuning process is complete, the fine-tuned model can be retrieved and used for inference.

Why Convert Data to JSONL Files?

JSONL (JSON Lines) is a convenient format for storing structured data where each line is a valid JSON object. This format is particularly useful for machine learning tasks for several reasons:

- **Line-by-Line Processing**: Each line in a JSONL file represents a separate JSON object, making it easy to process large datasets line by line without loading the entire file into memory.

- **Flexibility**: JSONL files can store complex nested structures, making them suitable for a wide range of data types.
- **Compatibility**: Many machine learning frameworks and tools, including OpenAI's API, support JSONL format, making it a standard choice for data interchange.

```
import os
import re
import json
import pandas as pd
from sklearn.model_selection import train_test_split

# Define file paths
movie_lines_file = "movie_lines.txt"
movie_conversations_file = "movie_conversations.txt"

# Step 1: Load the lines from movie_lines.txt
lines_dict = {}
with open(movie_lines_file, 'r', encoding='iso-8859-1') as file:
    for line in file:
        parts = line.strip().split(" +++$+++ ")
        if len(parts) == 5:  # Ensuring the correct format
            line_id, character_id, movie_id, character_name, text = parts
            lines_dict[line_id] = text

# Step 2: Load conversations from movie_conversations.txt
conversations = []
with open(movie_conversations_file, 'r', encoding='iso-8859-1') as file:
    for line in file:
        parts = line.strip().split(" +++$+++ ")
        if len(parts) == 4:  # Ensuring the correct format
            line_ids_str = parts[3]
            line_ids = re.findall(r"L[0-9]+", line_ids_str)  # Extracting
line IDs
            conversation = [lines_dict[line_id] for line_id in line_ids if
line_id in lines_dict]
            if len(conversation) > 1:  # Only add conversations with multiple
lines
                conversations.append(conversation)

# Prepare the data for training
jsonl_data = []
for conversation in conversations:
    jsonl_data.append({
        "messages": [
            {"role": "system", "content": "Engage in a conversation based on
movie dialogues."},
            {"role": "user", "content": conversation[0]},  # First line as
user input
            {"role": "assistant", "content": conversation[1]}  # Second line
as assistant response
        ]
    })

# Convert to DataFrame for easier handling
```

```
df = pd.DataFrame(jsonl_data)

# Split the data into training and validation sets (80% train, 20%
validation)
train_data, validation_data = train_test_split(df, test_size=0.2,
random_state=42)

def save_to_jsonl(data, output_file_path):
    # Save to JSONL format
    with open(output_file_path, 'w') as f:
        for _, row in data.iterrows():
            f.write(json.dumps(row.to_dict()) + '\n')

# Save the training and validation sets to separate JSONL files
train_output_file_path = 'movie_conversation_train.jsonl'
validation_output_file_path = 'movie_conversation_validation.jsonl'

save_to_jsonl(train_data, train_output_file_path)
save_to_jsonl(validation_data, validation_output_file_path)

print(f"Training dataset saved to {train_output_file_path}")
print(f"Validation dataset saved to {validation_output_file_path}")
```

Uploading Training and Validation Files for Fine-Tuning

In this section, we will upload the training and validation datasets to OpenAI's servers to prepare for fine-tuning the model. The datasets are in JSONL format, which is required for the fine-tuning process.

The steps involved are as follows:

1. **Upload Training File**: We use the `client.files.create()` method to upload the training dataset. The file is opened in binary read mode (`"rb"`) and the purpose is set to `"fine-tune"`.

2. **Upload Validation File**: Similarly, we upload the validation dataset using the same method.

3. **Print File Information**: After uploading, we print the information of both the training and validation files to confirm successful uploads.

```
train_file = client.files.create(
  file=open(train_output_file_path, "rb"),
  purpose="fine-tune"
)
```

```
valid_file = client.files.create(
  file=open(validation_output_file_path, "rb"),
  purpose="fine-tune"
)

print(f"Training file Info: {train_file}")
print(f"Validation file Info: {valid_file}")
```

Creating a Fine-Tuning Job

In this section, we will create a fine-tuning job using the OpenAI API. The fine-tuning job will use the training and validation datasets we previously uploaded. We will specify the model to be fine-tuned and set the hyperparameters for the training process.

The steps involved are as follows:

1. Specify Training and Validation Files: We use the IDs of the training and validation files that were uploaded to OpenAI's servers.
2. Set Model and Hyperparameters: We specify the model to be fine-tuned (`gpt-4o-mini-2024-07-18`) and set the hyperparameters, including the number of epochs, batch size, and learning rate multiplier.
3. Create Fine-Tuning Job: We create the fine-tuning job using the `client.fine_tuning.jobs.create()` method.
4. Retrieve Job ID and Status: After creating the job, we retrieve the job ID and status to monitor the fine-tuning process.

```
model = client.fine_tuning.jobs.create(
  training_file=train_file.id,
  validation_file=valid_file.id,
  model="gpt-4o-mini-2024-07-18",
  hyperparameters={
    "n_epochs": 3,
  "batch_size": 3,
  "learning_rate_multiplier": 0.3
  }
)
job_id = model.id
status = model.status

print(f'Fine-tuning model with jobID: {job_id}.')
print(f"Training Response: {model}")
```

```
print(f"Training Status: {status}")


import time
job = client.fine_tuning.jobs.retrieve("ftjob-nJMvXlFf73BUJKCrX7jkpuLO")

# Display the job details in a readable format
print(f"Job ID: {job.id}")
print(f"Model: {job.model}")
print(f"Fine-Tuned Model: {job.fine_tuned_model}")
print(f"Status: {job.status}")
print(f"Created At: {time.strftime('%Y-%m-%d %H:%M:%S',
time.gmtime(job.created_at))}")
print(f"Finished At: {time.strftime('%Y-%m-%d %H:%M:%S',
time.gmtime(job.finished_at))}")
print(f"Training File: {job.training_file}")
print(f"Validation File: {job.validation_file}")
print(f"Hyperparameters: {job.hyperparameters}")
print(f"Trained Tokens: {job.trained_tokens}")
print(f"Result Files: {job.result_files}")
print(f"Error: {job.error.message if job.error.code else 'None'}")


import time

# Function to retrieve and print fine-tuning job details including loss
def check_fine_tuning_job_status(client, job_id):
    while True:
        # Retrieve the state of the fine-tuning job
        job = client.fine_tuning.jobs.retrieve(job_id)

        # Print relevant details
        print(f"Status: {job.status}")
        print(f"Hyperparameters: {job.hyperparameters}")
        print(f"Model: {job.model}")
        print(f"Training File: {job.training_file}")
        print(f"Validation File: {job.validation_file}")

        # Check for error status
        if job.error.code is not None:
            print(f"Error: {job.error.message}")
            break

        # Check if the job has finished
        if job.status in ['succeeded', 'failed', 'cancelled']:
            print("Fine-tuning job has completed.")
            break

        # Optionally retrieve training metrics like loss
        if job.status in ['training', 'validating_files']:  # Check while in
training
            if hasattr(job, 'loss'):
                print(f"Current Loss: {job.loss}")  # Adjust based on actual
API response structure

        # Wait for a while before checking again
        print("Waiting for the job to finish...")
```

```
        time.sleep(10)  # Check every 10 seconds

job_id = 'ftjob-nJMvXlFf73BUJKCrX7jkpuLO'
check_fine_tuning_job_status(client, job_id)

print(f"Status: {job.status}")
print(f"Hyperparameters: {job.hyperparameters}")
print(f"Model: {job.model}")


import time
import requests

# Function to retrieve and print checkpoints for the fine-tuning job
def check_fine_tuning_job_checkpoints(job_id):
    headers = {
        "Authorization": f"Bearer {OPENAI_API_KEY}"
    }

    while True:
        # Call the OpenAI API to retrieve checkpoints
        response =
requests.get(f"https://api.openai.com/v1/fine_tuning/jobs/{job_id}/checkpoint
s", headers=headers)

        if response.status_code == 200:
            data = response.json()
            print(data)
            # Process and print each checkpoint
            for checkpoint in data['data']:
                step_number = checkpoint['step_number']
                train_loss = checkpoint['metrics'].get('train_loss')
                full_valid_mean_token_accuracy =
checkpoint['metrics'].get('train_mean_token_accuracy')
                fine_tuned_model_checkpoint =
checkpoint.get('fine_tuned_model_checkpoint')
                # Print relevant details for each checkpoint
                print(f"Step Number: {step_number}")
                print(f"Train Loss: {train_loss}")
                print(f"Mean Token Accuracy:
{full_valid_mean_token_accuracy}")

                print(f"CheckPoint Name: {fine_tuned_model_checkpoint}")
                print("------")

            # Check if there are more checkpoints to retrieve
            if not data.get("has_more", False):
                print("No more checkpoints to retrieve.")
                break
        else:
            print(f"Error retrieving checkpoints: {response.status_code},
{response.text}")
            break

        # Wait before checking again
        print("Waiting for the next checkpoint...")
        time.sleep(10)  # Adjust the wait time as needed
```

```
# Call the function to start checking for checkpoints
check_fine_tuning_job_checkpoints(job_id)


import openai

# Retrieve the list of fine-tuning jobs using the client and get the fine-
tuned model
result = client.fine_tuning.jobs.list()
fine_tuned_model = "ft:gpt-4o-mini-2024-07-18:personal::AJWkdy4H"  # Use the
correct key to get fine-tuned model name
print(f"Using Fine-Tuned Model: {fine_tuned_model}")

# Initialize the conversation history
messages = [
    {"role": "system", "content": "Given a social media post, classify
whether it indicates 'stress' or 'non-stress'."}
]

# Chat loop with the fine-tuned model
while True:
    # User input prompt
    user_input = input("You: ")

    # Check for exit condition
    if user_input.lower() in ["exit", "quit", "q"]:
        print("Ending the chat. Goodbye!")
        break

    # Append user message to conversation history
    messages.append({"role": "user", "content": user_input})

    # Call the OpenAI chat completion API using the client object
    response = client.chat.completions.create(
        model=fine_tuned_model,
        messages=messages
    )
    # Retrieve and display the model's response
    assistant_message = response.choices[0].message.content
    print(f"You: {user_input}")
    print(f"Assistant: {assistant_message}")


    # Append the assistant's response to conversation history
    messages.append({"role": "assistant", "content": assistant_message})
```

Appendix C – Flan-T5 Sample Chats

Sample 1:

```
Using GPU: Tesla T4
Using device: cuda
I'm a FLAN-T5 model! I will talk to you till you say 'bye' :)
You: What a beautiful day
Chatbot: I had a lovely day. The sun was shining and the weather was
beautiful.
You: What did you do today?
Chatbot: I went to the beach.
You: Which beach?
Chatbot: The beach at the north end of town
You: How far it is from here?
Chatbot: 3 miles
You: Did you eat anything at beach?
Chatbot: I didn't eat anything
You: What is your favorite dish?
Chatbot: Potato salad
You: How to prepare potato salad?
Chatbot: To prepare potato salad, first cut the potatoes in half lengthwise
and place them on a cutting board lined with foil.
You: Should we add tomato?
Chatbot: I don't think so.
You: Shall we go for a drive?
Chatbot: I don't think so.
You: Bye
Ending conversation. Goodbye!
```

Sample 2:

```
Using GPU: Tesla T4
Using device: cuda
I'm a FLAN-T5 model! I will talk to you till you say 'bye' :)
You: What is your name?
Chatbot: samuel
You: Where are you from?
Chatbot: England
You: Where do you live?
Chatbot: england
You: What do you do?
Chatbot: Actor
You: Which is your recent movie?
Chatbot: robin wayne
You: Was it a thriller?
Chatbot: yes
You: Does the hero survive in the climax battle?
Chatbot: yes
You: Do you watch movies?
Chatbot: yes
You: Which movie did you see recently?
Chatbot: A Beautiful Mind
You: Shall we go for a coffee?
Chatbot: I'd like a coffee.
You: Bye
Ending conversation. Goodbye!
```

Sample 3:

```
Using GPU: Tesla T4
Using device: cuda
I'm a FLAN-T5 model! I will talk to you till you say 'bye' :)
You: Hey, how’s your day going?
Chatbot: Hey, how are you?
You: All fine. Where did you go today?
Chatbot: I went to a concert.
You: Where was it?
Chatbot: New York City
You: Who played in the concert?
Chatbot: stevie wonder
You: What songs did he play?
Chatbot: "Let's Get Crazy"
You: How was the weather?
Chatbot: rainy
You: How long did the concert go?
Chatbot: 2 hours
You: Was it crowded?
Chatbot: yes
You: Bye
Ending conversation. Goodbye!
```