

# FLAN-T5-BASE CHATBOT MODEL

This model is used to provide textual answers to general questions. For this project we used the conversations dataset:

<https://www.kaggle.com/datasets/kreeshrajani/3k-conversations-dataset-for-chatbot>

## Import all the necessary libraries

```
In [ ]: import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from collections import Counter
import re
from nltk.corpus import stopwords
stop_words = set(stopwords.words('english'))
import nltk
from nltk.corpus import wordnet
from nltk.stem import WordNetLemmatizer
from sklearn.model_selection import train_test_split
from datasets import Dataset
from datasets import DatasetDict
import torch
from transformers import AutoModelForSeq2SeqLM, AutoTokenizer, GenerationConfig, TrainingArguments, Trainer
from peft import LoraConfig, get_peft_model, TaskType
from peft import PeftModel
import json
```

```
c:\Users\Paul\anaconda3\envs\torch\lib\site-packages\tqdm\auto.py:21: TqdmWarning: IPProgress not found. Please update
jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
from .autonotebook import tqdm as notebook_tqdm
```

## Read the data

We checked the structure of the dataset and counted the number of available questions/answers.

```
In [ ]: # Create a DataFrame
chatbot_df = pd.read_csv('Conversations_clean.csv')
```

```
In [ ]: # Show the first few rows of the DataFrame
chatbot_df.head()
```

```
Out[ ]:
```

	question	answer
0	hi, how are you doing?	i'm fine. how about yourself?
1	i'm fine. how about yourself?	i'm pretty good. thanks for asking.
2	i'm pretty good. thanks for asking.	no problem. so how have you been?
3	no problem. so how have you been?	i've been great. what about you?
4	i've been great. what about you?	i've been good. i'm in school right now.

```
In [ ]: chatbot_df.shape # print the dataset shape
```

```
Out[ ]: (3725, 2)
```

## EDA

EDA was performed by following a similar procedure to the one described by Singh (n.d.).

References: Singh, H. (n.d.). Complete Guide to EDA on Text Data. Kaggle. <https://www.kaggle.com/code/harshsingh2209/complete-guide-to-eda-on-text-data>

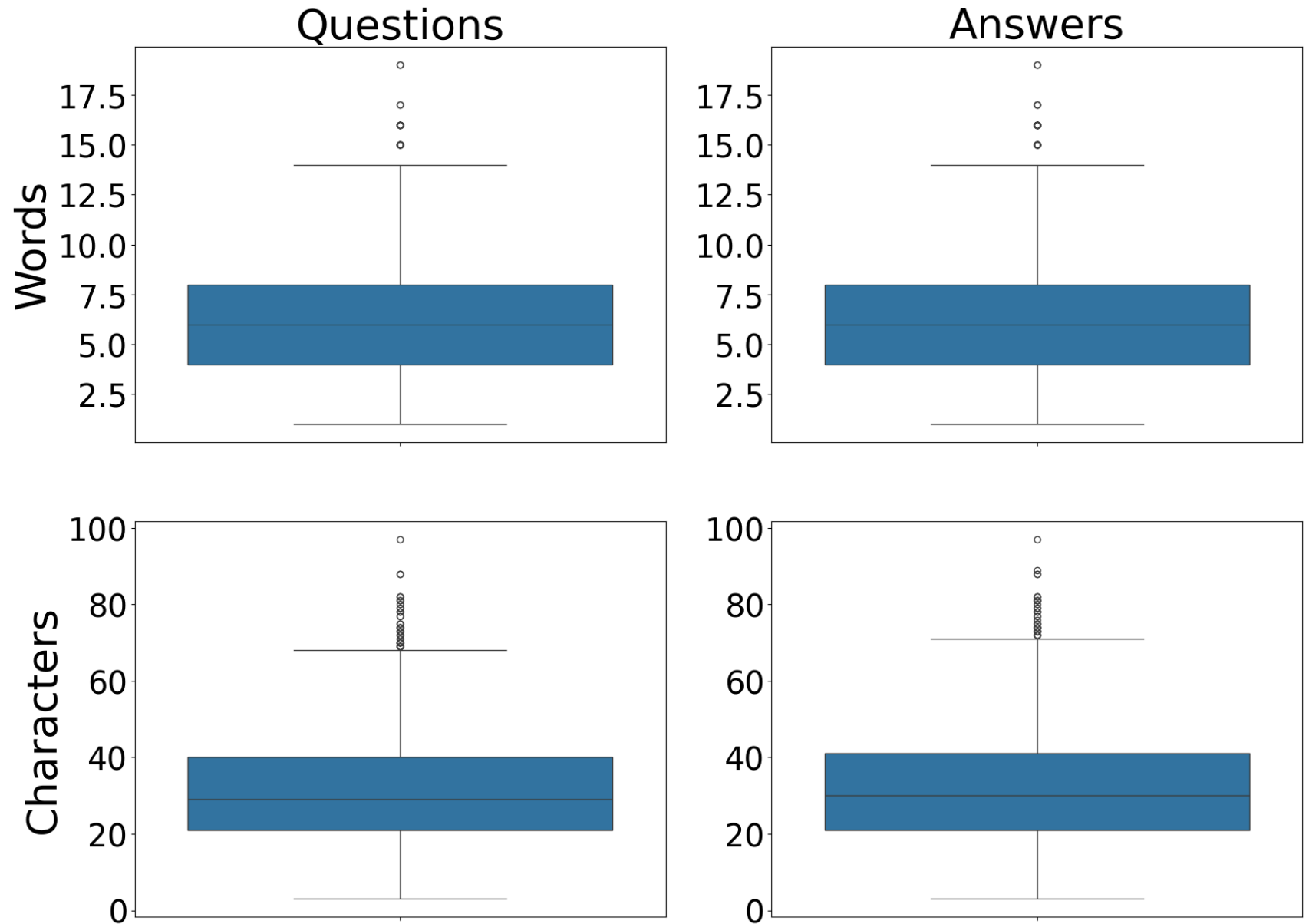
```
In [ ]: # Function to count the number of words in a sentence
def count_words(text):
    words = text.split() # extract the words from text
    num_words = len(words) # count the number of words
    return num_words

# Function to count the number of characters in a sentence
def count_characters(text):
    num_char = len(text) # count the number of chatacters
```

```
    return num_char

num_words_question = chatbot_df['question'].apply(count_words)
num_words_answer = chatbot_df['answer'].apply(count_words)
num_char_question = chatbot_df['question'].apply(count_characters)
num_char_answer = chatbot_df['answer'].apply(count_characters)

plt.figure(figsize = (20, 15))
plt.subplot(2, 2, 1)
sns.boxplot(y = num_words_question)
plt.yticks(fontsize = 30)
plt.ylabel('Words', fontsize = 40)
plt.title('Questions', fontsize = 40)
plt.subplot(2, 2, 2)
sns.boxplot(y = num_words_answer)
plt.yticks(fontsize = 30)
plt.ylabel('')
plt.title('Answers', fontsize = 40)
plt.subplot(2, 2, 3)
sns.boxplot(y = num_char_question)
plt.yticks(fontsize = 30)
plt.ylabel('Characters', fontsize = 40)
plt.subplot(2, 2, 4)
sns.boxplot(y = num_char_answer)
plt.yticks(fontsize = 30)
plt.ylabel('')
plt.savefig('Boxplots_QA.png')
plt.show()
```



```
In [ ]: # Identify the most frequent words in the question/answer columns
```

```
# Function to get the list of words in a sentence
```

```
def list_words(text):  
    words = text.split() # extract the words from text
```

```

    return words

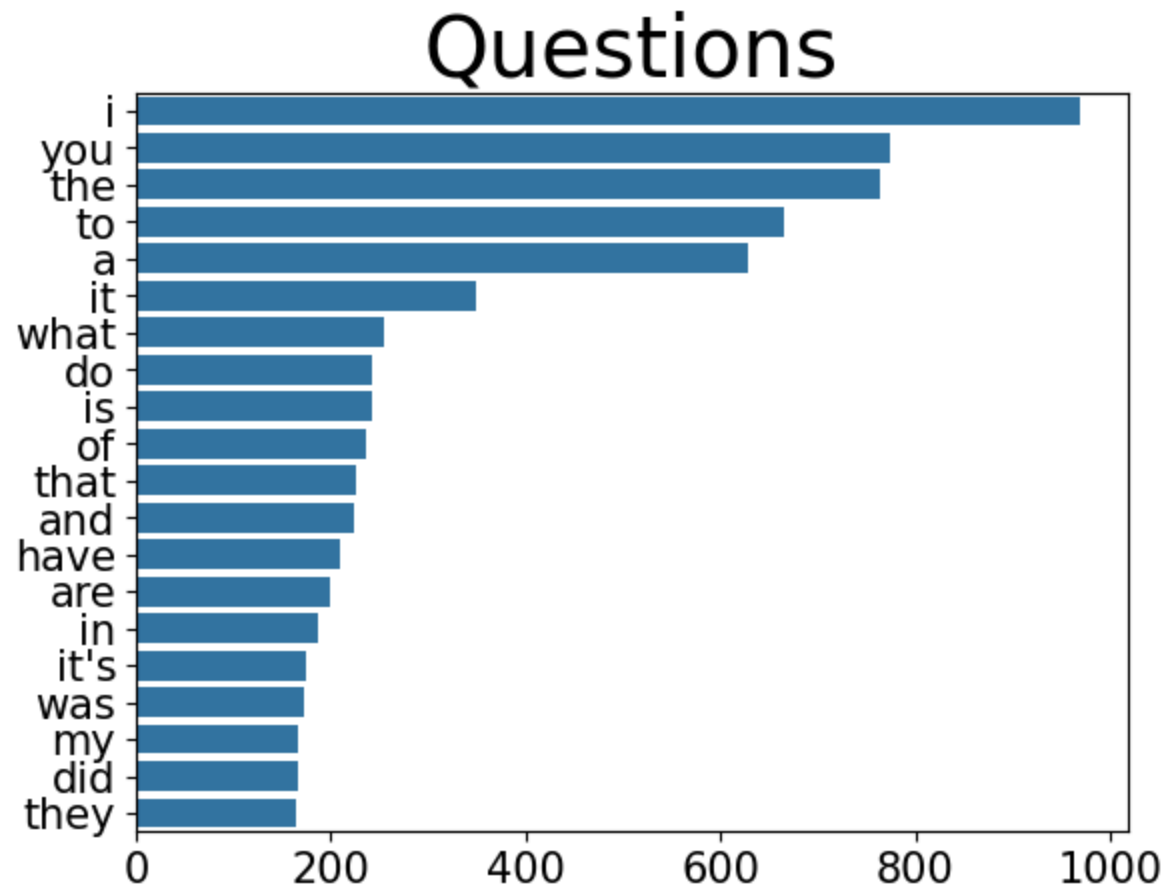
# Function to get the top 20 most common words and their counts
def words_freq(corpus):
    mostcommon = Counter(corpus).most_common(20)
    words = []
    counts = []
    for word, count in mostcommon:
        words.append(word)
        counts.append(count)
    return words, counts

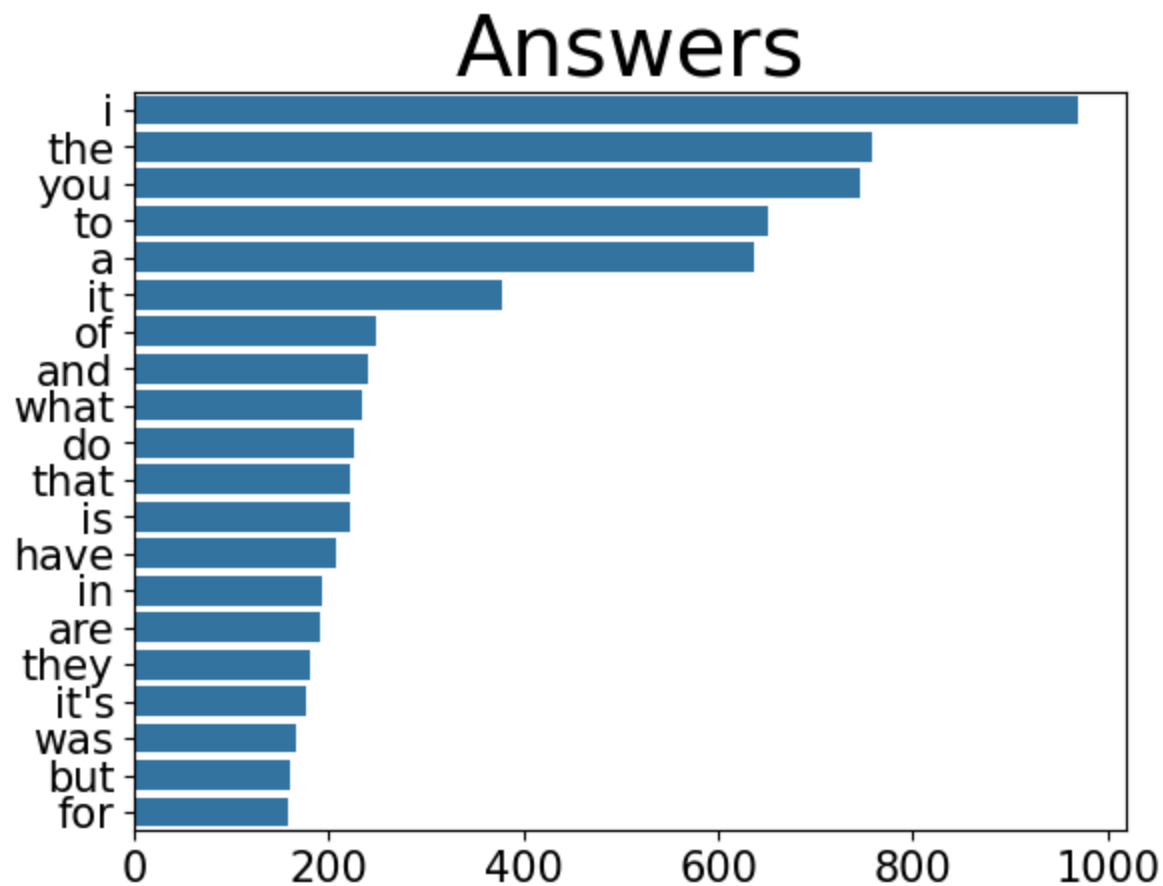
words_question = chatbot_df['question'].apply(list_words)
words_answer = chatbot_df['answer'].apply(list_words)
corpus_question = []
for jj in range(len(words_question)):
    corpus_question += words_question[jj] # all the words in all the questions
corpus_answer = []
for jj in range(len(words_answer)):
    corpus_answer += words_answer[jj] # all the words in all the answers
words_question, counts_question = words_freq(corpus_question) # top 20 most common words and their counts in question
words_answer, counts_answer = words_freq(corpus_answer) # top 20 most common words and their counts in answers

sns.barplot(x = counts_question, y = words_question)
plt.xticks(fontsize = 15)
plt.yticks(fontsize = 15)
plt.title('Questions', fontsize = 30)
plt.savefig('Common_Words_Questions.png')
plt.show()

sns.barplot(x = counts_answer, y = words_answer)
plt.xticks(fontsize = 15)
plt.yticks(fontsize = 15)
plt.title('Answers', fontsize = 30)
plt.savefig('Common_Words_Answers.png')
plt.show()

```





## Preprocessing

In order to replace contractions, we adopted the same procedure described in Replace apostrophe/short words in python (n.d.).

References: Replace apostrophe/short words in python. (n.d.). Stack Overflow. Retrieved July 24, 2024, from <https://stackoverflow.com/questions/43018030/replace-apostrophe-short-words-in-python>

```
In [ ]: # List of common contractions
contractions = {
    "ain't": "am not / are not",
    "aren't": "are not / am not",
```

```
"can't": "cannot",
"can't've": "cannot have",
"'cause": "because",
"could've": "could have",
"couldn't": "could not",
"couldn't've": "could not have",
"didn't": "did not",
"doesn't": "does not",
"don't": "do not",
"hadn't": "had not",
"hadn't've": "had not have",
"hasn't": "has not",
"haven't": "have not",
"he'd": "he had / he would",
"he'd've": "he would have",
"he'll": "he shall / he will",
"he'll've": "he shall have / he will have",
"he's": "he has / he is",
"how'd": "how did",
"how'd'y": "how do you",
"how'll": "how will",
"how's": "how has / how is",
"i'd": "I had / I would",
"i'd've": "I would have",
"i'll": "I shall / I will",
"i'll've": "I shall have / I will have",
"i'm": "I am",
"i've": "I have",
"isn't": "is not",
"it'd": "it had / it would",
"it'd've": "it would have",
"it'll": "it shall / it will",
"it'll've": "it shall have / it will have",
"it's": "it has / it is",
"let's": "let us",
"ma'am": "madam",
"mayn't": "may not",
"might've": "might have",
"mightn't": "might not",
"mightn't've": "might not have",
"must've": "must have",
"mustn't": "must not",
```



```
"mustn't've": "must not have",
"needn't": "need not",
"needn't've": "need not have",
"o'clock": "of the clock",
"oughtn't": "ought not",
"oughtn't've": "ought not have",
"shan't": "shall not",
"sha'n't": "shall not",
"shan't've": "shall not have",
"she'd": "she had / she would",
"she'd've": "she would have",
"she'll": "she shall / she will",
"she'll've": "she shall have / she will have",
"she's": "she has / she is",
"should've": "should have",
"shouldn't": "should not",
"shouldn't've": "should not have",
"so've": "so have",
"so's": "so as / so is",
"that'd": "that would / that had",
"that'd've": "that would have",
"that's": "that has / that is",
"there'd": "there had / there would",
"there'd've": "there would have",
"there's": "there has / there is",
"they'd": "they had / they would",
"they'd've": "they would have",
"they'll": "they shall / they will",
"they'll've": "they shall have / they will have",
"they're": "they are",
"they've": "they have",
"to've": "to have",
"wasn't": "was not",
"we'd": "we had / we would",
"we'd've": "we would have",
"we'll": "we will",
"we'll've": "we will have",
"we're": "we are",
"we've": "we have",
"weren't": "were not",
"what'll": "what shall / what will",
"what'll've": "what shall have / what will have",
```

```

"what're": "what are",
"what's": "what has / what is",
"what've": "what have",
"when's": "when has / when is",
"when've": "when have",
"where'd": "where did",
"where's": "where has / where is",
"where've": "where have",
"who'll": "who shall / who will",
"who'll've": "who shall have / who will have",
"who's": "who has / who is",
"who've": "who have",
"why's": "why has / why is",
"why've": "why have",
"will've": "will have",
"won't": "will not",
"won't've": "will not have",
"would've": "would have",
"wouldn't": "would not",
"wouldn't've": "would not have",
"y'all": "you all",
"y'all'd": "you all would",
"y'all'd've": "you all would have",
"y'all're": "you all are",
"y'all've": "you all have",
"you'd": "you had / you would",
"you'd've": "you would have",
"you'll": "you shall / you will",
"you'll've": "you shall have / you will have",
"you're": "you are",
"you've": "you have"
}

# Function to replace contractions, remove punctuation and apply lowercase
def clear_text(text):
    for word in text.split(): # remove contractions and apply lowercase
        word = word.lower()
        if word in contractions:
            text = text.replace(word, contractions[word])
    text = re.sub(r'^\w\s', '', text) # remove punctuation
    return text

```

```
chatbot_df['question'] = chatbot_df['question'].apply(clear_text)
chatbot_df['answer'] = chatbot_df['answer'].apply(clear_text)
chatbot_df.head()
```

Out [ ]:

	question	answer
0	hi how are you doing	I am fine how about yourself
1	I am fine how about yourself	I am pretty good thanks for asking
2	I am pretty good thanks for asking	no problem so how have you been
3	no problem so how have you been	I have been great what about you
4	I have been great what about you	I have been good I am in school right now

In [ ]:

```
# Tokenization
chatbot_df['token_question'] = chatbot_df['question'].apply(nltk.word_tokenize)
chatbot_df['token_answer'] = chatbot_df['answer'].apply(nltk.word_tokenize)
chatbot_df.head()
```

Out [ ]:

	question	answer	token_question	token_answer
0	hi how are you doing	I am fine how about yourself	[hi, how, are, you, doing]	[I, am, fine, how, about, yourself]
1	I am fine how about yourself	I am pretty good thanks for asking	[I, am, fine, how, about, yourself]	[I, am, pretty, good, thanks, for, asking]
2	I am pretty good thanks for asking	no problem so how have you been	[I, am, pretty, good, thanks, for, asking]	[no, problem, so, how, have, you, been]
3	no problem so how have you been	I have been great what about you	[no, problem, so, how, have, you, been]	[I, have, been, great, what, about, you]
4	I have been great what about you	I have been good I am in school right now	[I, have, been, great, what, about, you]	[I, have, been, good, I, am, in, school, right...]

In [ ]:

```
# Lemmatization

# Function to obtain the right positional tagging prior to Lemmatization
# Same function presented in Python - Lemmatization Approaches with Examples (n.d.)
def right_pos_tagging(tag):
```

```

if tag.startswith('J'):
    return wordnet.ADJ
elif tag.startswith('V'):
    return wordnet.VERB
elif tag.startswith('N'):
    return wordnet.NOUN
elif tag.startswith('R'):
    return wordnet.ADV
else:
    return None

# Obtain the right positional tagging prior to Lemmatization
# Positional tags were modified as suggested by Python - Lemmatization Approaches with Examples (n.d.)

tokens = chatbot_df['token_question']
new_tag_tokens = []
for jj in range(len(tokens)):
    tokens_jj = tokens[jj] # tokens at the jjth row
    tag_tokens_jj = nltk.pos_tag(tokens_jj) # POS tags for the generic tokens_jj
    new_tag_tokens.append(list(map(lambda x: (x[0], right_pos_tagging(x[1])), tag_tokens_jj))) # modified POS tags for
tagged_token_question = new_tag_tokens

tokens = chatbot_df['token_answer']
new_tag_tokens = []
for jj in range(len(tokens)):
    tokens_jj = tokens[jj] # tokens at the jjth row
    tag_tokens_jj = nltk.pos_tag(tokens_jj) # POS tags for the generic tokens_jj
    new_tag_tokens.append(list(map(lambda x: (x[0], right_pos_tagging(x[1])), tag_tokens_jj))) # modified POS tags for
tagged_token_answer = new_tag_tokens

# Implement Lemmatization on the tokens
# A procedure similar to the one described in Python - Lemmatization Approaches with Examples (n.d.) and by Kumar (2018)

wnl = WordNetLemmatizer()
Lemmatization = []
for jj in range(len(tagged_token_question)):
    lemmatized_question = []
    # Same lines of codes used in Python - Lemmatization Approaches with Examples (n.d.)
    for word, tag in tagged_token_question[jj]:
        if tag is None:
            lemmatized_question.append(word)
        else:

```

```

        lemmatized_question.append(wnl.lemmatize(word, tag))
    Lemmatization.append(lemmatized_question)
chatbot_df['lem_question'] = Lemmatization

Lemmatization = []
for jj in range(len(tagged_token_answer)):
    lemmatized_answer = []
    # Same lines of codes used in Python - Lemmatization Approaches with Examples (n.d.)
    for word, tag in tagged_token_answer[jj]:
        if tag is None:
            lemmatized_answer.append(word)
        else:
            lemmatized_answer.append(wnl.lemmatize(word, tag))
    Lemmatization.append(lemmatized_answer)
chatbot_df['lem_answer'] = Lemmatization

chatbot_df.head()

# References
# Kumar, R. (2021, August 6). Natural Language Processing | Text Preprocessing | Spacy vs NLTK. Medium. https://medium.com/@rkumar1998/natural-language-processing-text-preprocessing-spacy-vs-nltk-1234567890
# Python - Lemmatization Approaches with Examples. (n.d.). Geeks for Geeks. https://www.geeksforgeeks.org/python-lemmatization-approaches-with-examples/

```

Out[ ]:

	question	answer	token_question	token_answer	lem_question	lem_answer
0	hi how are you doing	I am fine how about yourself	[hi, how, are, you, doing]	[I, am, fine, how, about, yourself]	[hi, how, be, you, do]	[I, be, fine, how, about, yourself]
1	I am fine how about yourself	I am pretty good thanks for asking	[I, am, fine, how, about, yourself]	[I, am, pretty, good, thanks, for, asking]	[I, be, fine, how, about, yourself]	[I, be, pretty, good, thanks, for, ask]
2	I am pretty good thanks for asking	no problem so how have you been	[I, am, pretty, good, thanks, for, asking]	[no, problem, so, how, have, you, been]	[I, be, pretty, good, thanks, for, ask]	[no, problem, so, how, have, you, be]
3	no problem so how have you been	I have been great what about you	[no, problem, so, how, have, you, been]	[I, have, been, great, what, about, you]	[no, problem, so, how, have, you, be]	[I, have, be, great, what, about, you]
4	I have been great what about you	I have been good I am in school right now	[I, have, been, great, what, about, you]	[I, have, been, good, I, am, in, school, right...]	[I, have, be, great, what, about, you]	[I, have, be, good, I, be, in, school, right, ...]

# Flan-T5-Base Fine Tuning

This section was created by following the same procedure described by Bhandare (n.d.).

References: Bhandare, A. (n.d.). Fine-tune Flan-T5-base for chat with PEFT/LoRA! Kaggle.

<https://www.kaggle.com/code/ajinkyabhandare2002/fine-tune-flan-t5-base-for-chat-with-peft-lora>

## Import tokenizer and model

```
In [ ]: model_name = 'google/flan-t5-base' # define the model to use
model = AutoModelForSeq2SeqLM.from_pretrained(model_name, torch_dtype = torch.bfloat16) # Load the model with an indi
tokenizer = AutoTokenizer.from_pretrained(model_name) # Load the tokenizer
```

## Preprocess data for retrain

```
In [ ]: chatbot_df = chatbot_df.drop(columns = ['token_question', 'token_answer', 'lem_question', 'lem_answer']) # just use the
```

```
In [ ]: # Define the training, validation and testing sets
train_data, val_test_data = train_test_split(chatbot_df, test_size = 0.2, random_state = 42)
val_data, test_data = train_test_split(val_test_data, test_size = 0.5, random_state = 42)
train_dataset = Dataset.from_pandas(train_data)
val_dataset = Dataset.from_pandas(val_data)
test_dataset = Dataset.from_pandas(test_data)
```

```
In [ ]: # Create a working dataset, including the training, validation and testing sets
working_dataset = DatasetDict({
    "train": train_dataset,
    "validation": val_dataset,
    "test": test_dataset,
})
working_dataset
```

```
Out[ ]: DatasetDict({
  train: Dataset({
    features: ['question', 'answer', '__index_level_0__'],
    num_rows: 2980
  })
  validation: Dataset({
    features: ['question', 'answer', '__index_level_0__'],
    num_rows: 372
  })
  test: Dataset({
    features: ['question', 'answer', '__index_level_0__'],
    num_rows: 373
  })
})
```

```
In [ ]: # Remove the unnecessary columns
working_dataset["train"] = working_dataset["train"].remove_columns("__index_level_0__")
working_dataset["validation"] = working_dataset["validation"].remove_columns("__index_level_0__")
working_dataset["test"] = working_dataset["test"].remove_columns("__index_level_0__")
working_dataset
```

```
Out[ ]: DatasetDict({
  train: Dataset({
    features: ['question', 'answer'],
    num_rows: 2980
  })
  validation: Dataset({
    features: ['question', 'answer'],
    num_rows: 372
  })
  test: Dataset({
    features: ['question', 'answer'],
    num_rows: 373
  })
})
```

```
In [ ]: # Function to tokenize questions/answers and get their input_ids
def tok_fun(example):
    example['input_ids'] = tokenizer(example['question'], padding = 'max_length', truncation = True, return_tensors = 'pt')
    example['labels'] = tokenizer(example["answer"], padding = 'max_length', truncation = True, return_tensors = "pt")
    return example
```

```
tokenized_datasets = working_dataset.map(tok_fun, batched = True) # apply the function to tokenize to questions and answers
tokenized_datasets = tokenized_datasets.remove_columns(['question', 'answer']) # remove the original question and answer columns
tokenized_datasets
```

```
Map: 100%|██████████| 2980/2980 [00:00<00:00, 6435.89 examples/s]
Map: 100%|██████████| 372/372 [00:00<00:00, 6722.25 examples/s]
Map: 100%|██████████| 373/373 [00:00<00:00, 6957.27 examples/s]
```

```
Out[ ]: DatasetDict({
  train: Dataset({
    features: ['input_ids', 'labels'],
    num_rows: 2980
  })
  validation: Dataset({
    features: ['input_ids', 'labels'],
    num_rows: 372
  })
  test: Dataset({
    features: ['input_ids', 'labels'],
    num_rows: 373
  })
})
```

## Setup the PEFT/LoRA model

```
In [ ]: lora_config = LoraConfig(
    r = 8,
    lora_alpha = 8,
    target_modules = ["q", "v"],
    lora_dropout = 0.05,
    bias = "none",
    task_type = TaskType.SEQ_2_SEQ_LM
)
```

## Add the LoRA adapter layers/parameters to the LLM model

```
In [ ]: peft_model = get_peft_model(model, lora_config)
```

```
bin c:\Users\Paul\anaconda3\envs\torch\lib\site-packages\bitsandbytes\libbitsandbytes_cuda118.dll
```

## Prepare the trainer



```
In [ ]: # Define the output directory
output_dir = f'./peft-conversation-training'

# Define the training arguments
peft_training_args = TrainingArguments(
    output_dir = output_dir,
    auto_find_batch_size = True,
    learning_rate = 1e-3,
    num_train_epochs = 5,
    save_steps = 100,
    save_strategy = 'steps',
    evaluation_strategy = 'steps',
    eval_steps = 10,
)

# Create the trainer instance
peft_trainer = Trainer(
    model = peft_model,
    args = peft_training_args,
    train_dataset = tokenized_datasets["train"],
    eval_dataset = tokenized_datasets['validation']
)
```

c:\Users\Paul\anaconda3\envs\torch\lib\site-packages\transformers\training\_args.py:1525: FutureWarning: `evaluation\_strategy` is deprecated and will be removed in version 4.46 of 🤗 Transformers. Use `eval\_strategy` instead  
warnings.warn(

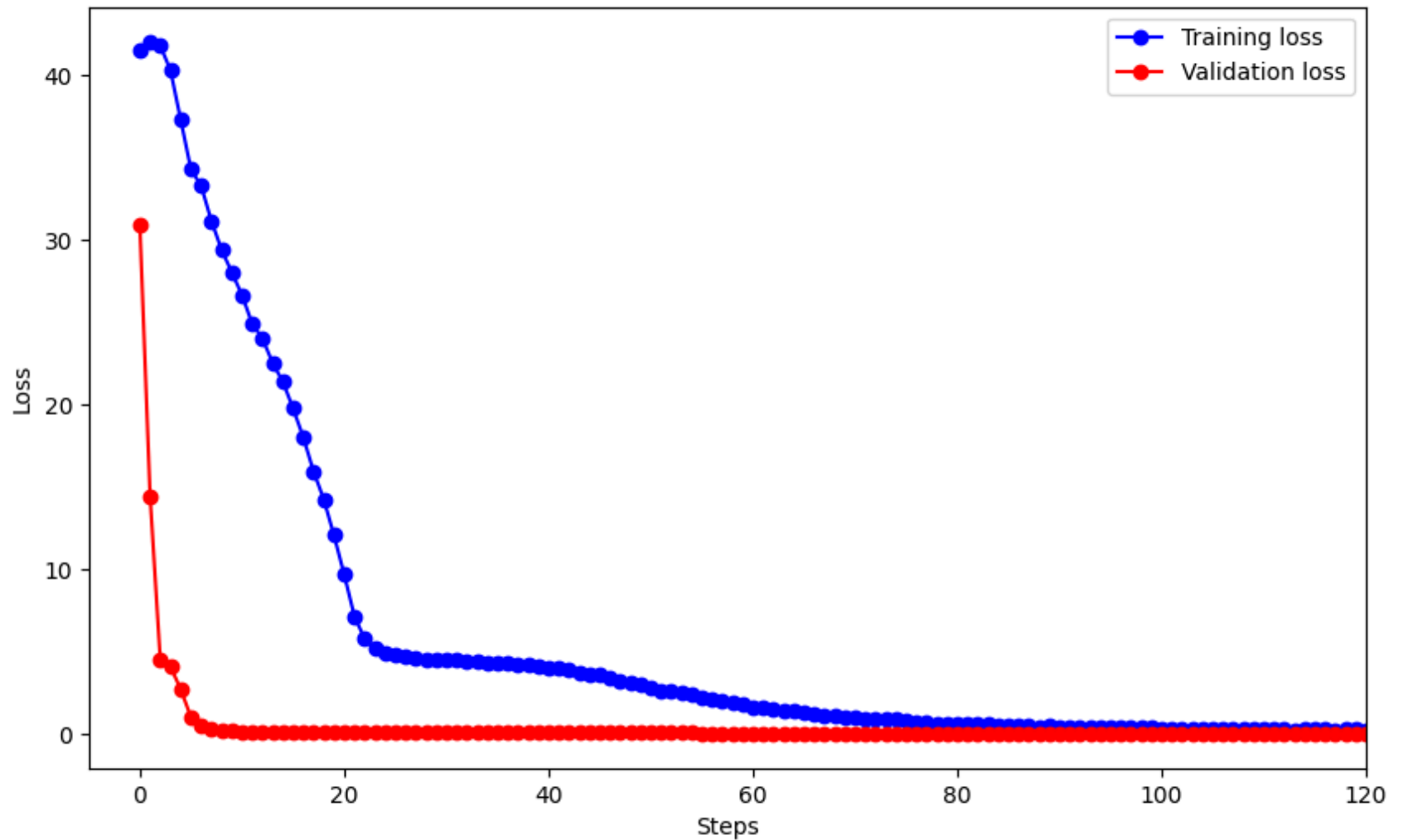
## Train the model

```
In [ ]: # Run the trainer
peft_trainer.train()
```

```
In [ ]: # Path to the log file
log_file_path = f"{output_dir}/checkpoint-1865/trainer_state_loss.json"
log_file_path_val = f"{output_dir}/checkpoint-1865/trainer_state_val.json"

# Load the log file
with open(log_file_path, "r") as log_file:
    logs = json.load(log_file)
with open(log_file_path_val, "r") as log_file_val:
```

```
logs_val = json.load(log_file_val)
# Extract the training losses
training_losses = [entry["loss"] for entry in logs["log_history"] if "loss" in entry]
validation_losses = [entry["eval_loss"] for entry in logs_val["log_history"] if "eval_loss" in entry]
# Plot the learning curve
plt.figure(figsize = (10, 6))
plt.plot(training_losses, 'bo-', label = "Training loss")
plt.plot(validation_losses, 'ro-', label = "Validation loss")
plt.xlabel("Steps")
plt.ylabel("Loss")
plt.xlim([-5, 120])
plt.legend()
plt.savefig('Learning_Curve_Chatbot')
plt.show()
```



### Save the fine-tuned model

```
In [ ]: peft_model_path="./peft-conversation-checkpoint-local"

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

### Define the final model

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

# Add the trained adapter to the original Flan-T5-Base model
peft_model = PeftModel.from_pretrained(peft_model_base,
                                       'peft-conversation-checkpoint-local',
                                       torch_dtype = torch.bfloat16,
                                       is_trainable = False)
```

## Evaluate the final model performance

Textual responses were generated as suggested by Bhandare (n.d.).

References: Bhandare, A. (n.d.). Fine-tune Flan-T5-base for chat with PEFT/LoRA! Kaggle.

<https://www.kaggle.com/code/ajinkyabhandare2002/fine-tune-flan-t5-base-for-chat-with-peft-lora>

```
In [ ]: def insert_prompt(prompt):

    input_ids = tokenizer(prompt, return_tensors = "pt").input_ids # get the input_ids from the received prompt
    outputs = peft_model.generate(input_ids = input_ids, generation_config = GenerationConfig(max_new_tokens = 200,
    text_output = tokenizer.decode(outputs[0], skip_special_tokens = True) # convert the model outputs into text

    return text_output
```

```
In [ ]: prompt = "Hello, how are you?"

print(f'question:\n{prompt}')
print(f'PEFT MODEL: {insert_prompt(prompt)}')
```

```
question:
Hello, how are you?
PEFT MODEL: I'm fine. I'm fine.
```

```
In [ ]: prompt = "What is your name?"

print(f'question:\n{prompt}')
print(f'PEFT MODEL: {insert_prompt(prompt)}')
```

question:  
What is your name?  
PEFT MODEL: samuel

```
In [ ]: prompt = "Are you happy today?"

print(f'question:\n{prompt}')
print(f'PEFT MODEL: {insert_prompt(prompt)}')
```

question:  
Are you happy today?  
PEFT MODEL: I am happy

```
In [ ]: prompt = "What is your favorite color?"

print(f'question:\n{prompt}')
print(f'PEFT MODEL: {insert_prompt(prompt)}')
```

question:  
What is your favorite color?  
PEFT MODEL: blue

```
In [ ]: prompt = "Do you think it will rain in the afternoon?"

print(f'question:\n{prompt}')
print(f'PEFT MODEL: {insert_prompt(prompt)}')
```

question:  
Do you think it will rain in the afternoon?  
PEFT MODEL: It is not expected to rain

```
In [ ]: prompt = "What is your dream?"

print(f'question:\n{prompt}')
print(f'PEFT MODEL: {insert_prompt(prompt)}')
```

question:  
What is your dream?  
PEFT MODEL: i want to be a good singer

```
In [ ]: prompt = "What is the capital of California?"
```

```
print(f'question:\n{prompt}')
```

```
print(f'PEFT MODEL: {insert_prompt(prompt)}')
```

question:

What is the capital of California?

PEFT MODEL: san francisco

```
In [ ]: prompt = "What is the capital of Italy?"
```

```
print(f'question:\n{prompt}')
```

```
print(f'PEFT MODEL: {insert_prompt(prompt)}')
```

question:

What is the capital of Italy?

PEFT MODEL: rome

```
In [ ]: prompt = "When does summer begin?"
```

```
print(f'question:\n{prompt}')
```

```
print(f'PEFT MODEL: {insert_prompt(prompt)}')
```

question:

When does summer begin?

PEFT MODEL: summer

```
In [ ]: prompt = "What is the best drink for a hangover?"
```

```
print(f'question:\n{prompt}')
```

```
print(f'PEFT MODEL: {insert_prompt(prompt)}')
```

question:

What is the best drink for a hangover?

PEFT MODEL: a stout

```
In [ ]: prompt = "What is the probability that two apples will fall at the same time?"
```

```
print(f'question:\n{prompt}')
```

```
print(f'PEFT MODEL: {insert_prompt(prompt)}')
```

question:

What is the probability that two apples will fall at the same time?

PEFT MODEL: 1 / 2

```
In [ ]: prompt = "Could you please recommend an American dish?"

print(f'question:\n{prompt}')
print(f'PEFT MODEL: {insert_prompt(prompt)}')
```

question:  
Could you please recommend an American dish?  
PEFT MODEL: a burger

```
In [ ]: prompt = "Where would you go on vacation this summer?"

print(f'question:\n{prompt}')
print(f'PEFT MODEL: {insert_prompt(prompt)}')
```

question:  
Where would you go on vacation this summer?  
PEFT MODEL: san francisco

```
In [ ]: prompt = "What is Buddhism?"

print(f'question:\n{prompt}')
print(f'PEFT MODEL: {insert_prompt(prompt)}')
```

question:  
What is Buddhism?  
PEFT MODEL: Buddhism is a philosophy

```
In [ ]: prompt = "Could you suggest a person's name with C?"

print(f'question:\n{prompt}')
print(f'PEFT MODEL: {insert_prompt(prompt)}')
```

question:  
Could you suggest a person's name with C?  
PEFT MODEL: charlie c

```
In [ ]: prompt = "Who is Chris Martin?"

print(f'question:\n{prompt}')
print(f'PEFT MODEL: {insert_prompt(prompt)}')
```

question:  
Who is Chris Martin?  
PEFT MODEL: british actor

```
In [ ]: prompt = "When will next Easter be celebrated?"

print(f'question:\n{prompt}')
print(f'PEFT MODEL: {insert_prompt(prompt)}')
```

question:  
When will next Easter be celebrated?  
PEFT MODEL: in the spring

```
In [ ]: prompt = "What is rosemary?"

print(f'question:\n{prompt}')
print(f'PEFT MODEL: {insert_prompt(prompt)}')
```

question:  
What is rosemary?  
PEFT MODEL: rosemary

```
In [ ]: prompt = "What would you do in case of dehydration?"

print(f'question:\n{prompt}')
print(f'PEFT MODEL: {insert_prompt(prompt)}')
```

question:  
What would you do in case of dehydration?  
PEFT MODEL: drink water