Preprocessing EDA and BART notebook

October 23, 2023

0.0.1 Introduction

The evolution of chatbots has revolutionized the way we interact with technology, providing a more natural and interactive means of communication. One of the most exciting developments in this domain has been the integration of artificial intelligence to create intelligent chatbots that can understand and respond to user queries in a conversational manner.

This project aims to design and implement a generative-based chatbot using state-of-the-art architectures and techniques. The primary objective is to build a chatbot that can carry out multi-turn conversations, adapt to context, and handle a variety of topics, with a focus on question-and-answer-style interactions.

The dataset chosen for this project is the Stanford Question Answering Dataset (SQuAD), a popular dataset in the natural language processing community. SQuAD consists of questions posed by crowdworkers on a set of Wikipedia articles, where the answer to every question is a segment of text from the corresponding reading passage.

For the model architecture, we have decided to utilize BART (Bidirectional and Auto-Regressive Transformers), a powerful transformer-based model known for its effectiveness in sequence-to-sequence tasks, including text generation and question-answering.

The steps we will follow to achieve our goal include:

1. Data Cleaning and Preparation
2. Exploratory Data Analysis
3. Model Selection
4. Model Training
5. Model Analysis
6. Conclusion and Evaluation

The final output of this project will be a fully functional chatbot capable of accurately responding to user queries in a conversational manner. Evaluation metrics such as BLEU Score and ROUGE Scores will be employed, with a focus on the SacreBLEU implementation, to quantitatively assess the performance of the chatbot.

The project report will detail the challenges faced, solutions implemented, model architecture, evaluation results, and user feedback, followed by potential future improvements and scalability options.

With this introduction in place, we are now ready to move on to Step 2: Data Cleaning and Preparation. Let me know if you're ready to proceed!

Data Cleaning and Preparation

In this step, we will perform the following tasks:

Load the Dataset: Load the SQuAD dataset files.

Data Inspection: Inspect the data to understand its structure and content. Data Cleaning:Clean and preprocess the data to make it suitable for training the model.

Data Transformation: Transform the data into a format that can be used to train the BART model.

Loading data:

[ ]:
```python
import json

# Load the SQuAD dataset files
file_paths = ['train-v1.1.json', 'dev-v1.1.json']

# Load and inspect the structure of the training dataset
with open(file_paths[0]) as file:
    train_data = json.load(file)

# Show the keys at the top level of the JSON structure
train_data.keys()
```

[ ]:
```
dict_keys(['data', 'version'])
```

Data Inspection:

[ ]:
```python
# Inspect the structure of the "data" key
train_data['data'][0].keys(), train_data['data'][0]['title'],␣
 ↪train_data['data'][0]['paragraphs'][0].keys()
```

[ ]:
```
(dict_keys(['title', 'paragraphs']),
 'University_of_Notre_Dame',
 dict_keys(['context', 'qas']))
```

The structure under the "data" key is as follows:

title: This key contains the title of the article or section.

paragraphs: This key contains a list of paragraphs, where each paragraph has the following structure:

- **context:** This key contains the text of the paragraph.
- **qas:** This key contains a list of question-answer pairs related to the paragraph, where each question-answer pair has the following structure:

1. **question**: The question text.
2. **answers**: A list of possible answers to the question, where each answer has the following structure:

- **text**: The answer text.
- **answer__start**: The starting index of the answer in the paragraph context.

Data Transformation:

First, we'll start by Extracting Question-Answer Pairs:

Here, we will Extract question-answer pairs and their corresponding contexts from the dataset.

[ ]:
```
# Extracting Question-Answer Pairs and Contexts

questions = []
answers = []
contexts = []

for article in train_data['data']:
    for paragraph in article['paragraphs']:
        context = paragraph['context']
        for qa in paragraph['qas']:
            question = qa['question']
            for answer in qa['answers']:
                questions.append(question)
                answers.append(answer['text'])
                contexts.append(context)

# Display the first few question-answer pairs and their contexts
questions[:5], answers[:5], contexts[:5]
```

We have successfully extracted the question-answer pairs and their corresponding contexts from the dataset.

Next, we will proceed with handling any missing values in the dataset. Let's check for any missing values in the extracted data.

[ ]:
```
# Check for missing values in the extracted data
missing_questions = sum(1 for q in questions if q is None or q == '')
missing_answers = sum(1 for a in answers if a is None or a == '')
missing_contexts = sum(1 for c in contexts if c is None or c == '')

missing_questions, missing_answers, missing_contexts
```

[ ]:
```
(0, 0, 0)
```

Looking at our results it seems that there are no missing values in the extracted data.

Next we will proceed with preprocessing the extracted data.

Preprocessing is the process of converting text into a format that can be easily processed by machine learning models. In this step, we will use a tokenizer suitable for the BART model to convert the text data into tokens.

Tokenization

For tokenization, we will use the tokenizer provided by the Hugging Face Transformers library, which is specifically designed for the BART model. The tokenizer will convert the text data into tokens, which are numerical representations of words or subwords. These tokens can then be used as input for the BART model.

[ ]:
```
from transformers import BartTokenizer

# Load the BART tokenizer
tokenizer = BartTokenizer.from_pretrained('facebook/bart-large')

# Tokenize the text data
inputs = tokenizer(questions, answers, max_length=512, truncation=True,␣
 ↪padding='max_length', return_tensors='pt')

# Display the tokenized data
print(inputs)
```

0.1 Exploratory Data Analysis

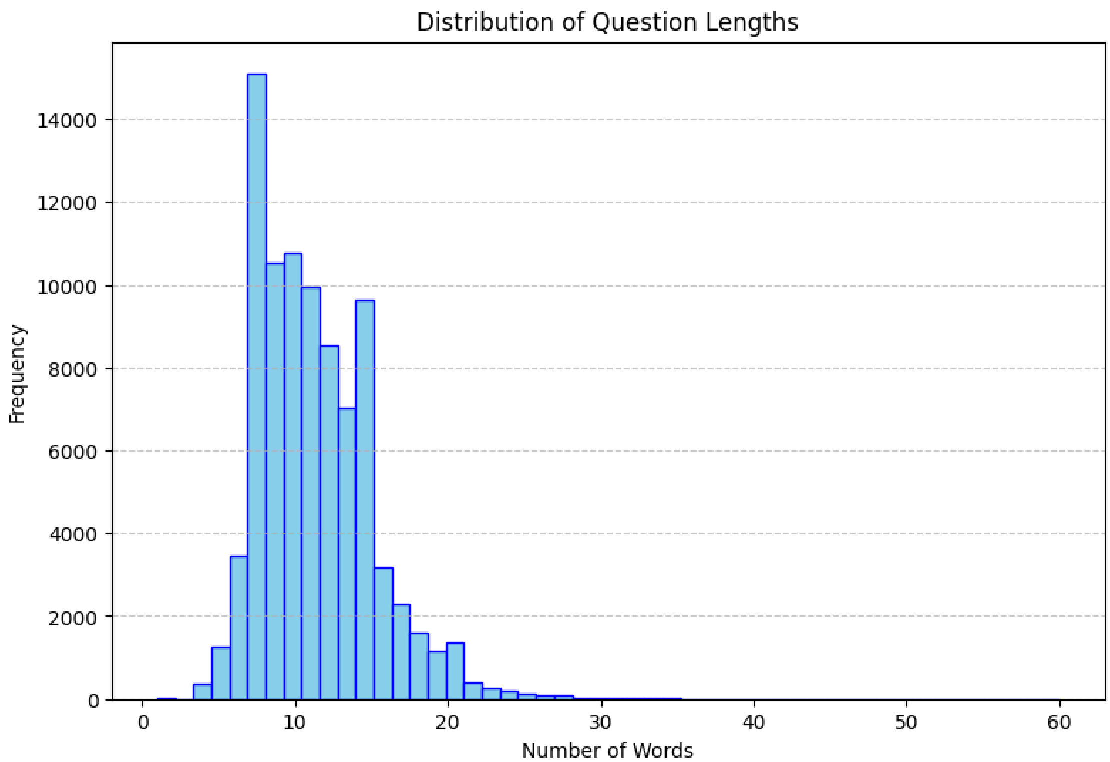
In this step, we will perform exploratory data analysis (EDA) to understand the characteristics of the dataset. EDA helps us to uncover patterns, insights, and relationships in the data, which can inform our model training and evaluation.

Distribution of Question Lengths

We'll start by analyzing the distribution of the number of words in the questions. This will give us an understanding of how concise or detailed the questions in the dataset are.

[ ]:
```
import nltk
import matplotlib.pyplot as plt
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords


# Download the NLTK data (if not already downloaded)
nltk.download('punkt')

# Tokenize the questions into words
question_words = [word_tokenize(q) for q in questions]

# Calculate the length of each question
question_lengths = [len(q) for q in question_words]

# Plot the distribution of question lengths
plt.figure(figsize=(9, 6))
plt.hist(question_lengths, bins=50, color='skyblue', edgecolor='blue')
plt.title('Distribution of Question Lengths')
plt.xlabel('Number of Words')
plt.ylabel('Frequency')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

```
[nltk_data] Downloading package punkt to C:\Users\Reed
[nltk_data]     Oken\AppData\Roaming\nltk_data…
[nltk_data]   Package punkt is already up-to-date!
```



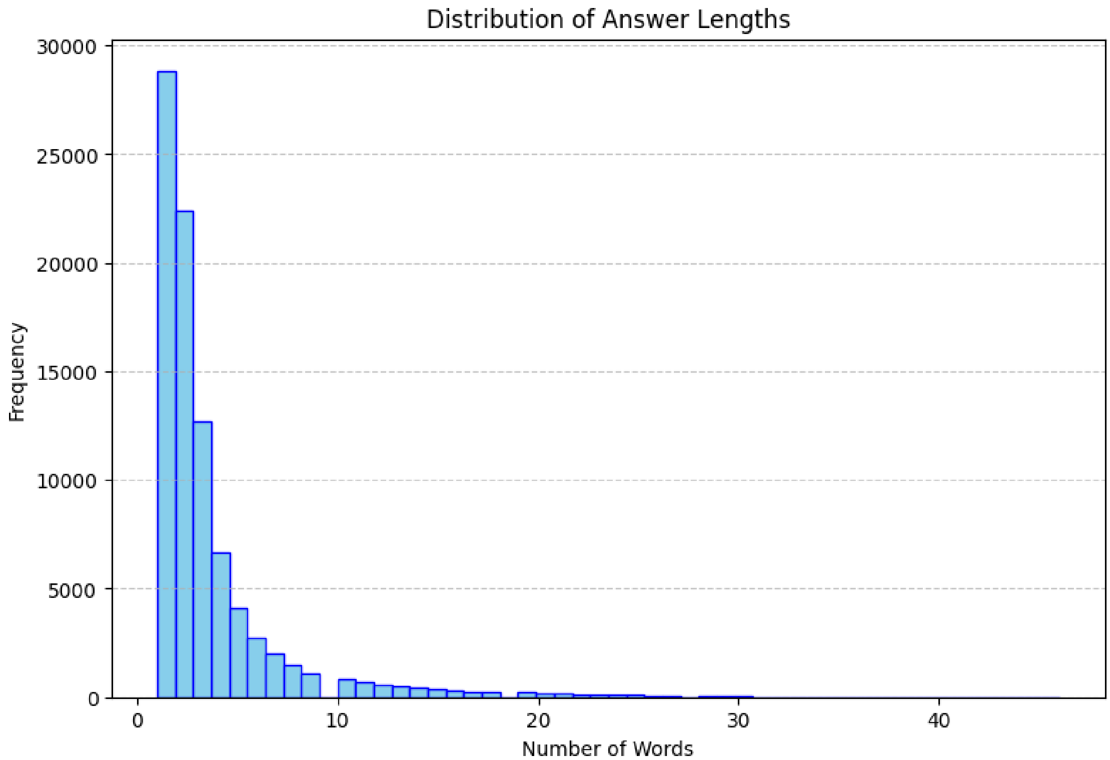
It looks like the majority of questions in the dataset have lengths between 8 to 14 words, with a peak at around 10 to 11 words. This indicates that most questions are fairly concise.

Distribution of Answer Lengths

Next, we'll analyze the distribution of the number of words in the answers. This will give us an understanding of how concise or detailed the answers in the dataset are. You can use the following code to perform this analysis:

```
[ ]: # Tokenize the answers into words
answer_words = [word_tokenize(a) for a in answers]

# Calculate the length of each answer
answer_lengths = [len(a) for a in answer_words]

# Plot the distribution of answer lengths
plt.figure(figsize=(9, 6))
plt.hist(answer_lengths, bins=50, color='skyblue', edgecolor='blue')
plt.title('Distribution of Answer Lengths')
plt.xlabel('Number of Words')
```

```
plt.ylabel('Frequency')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```



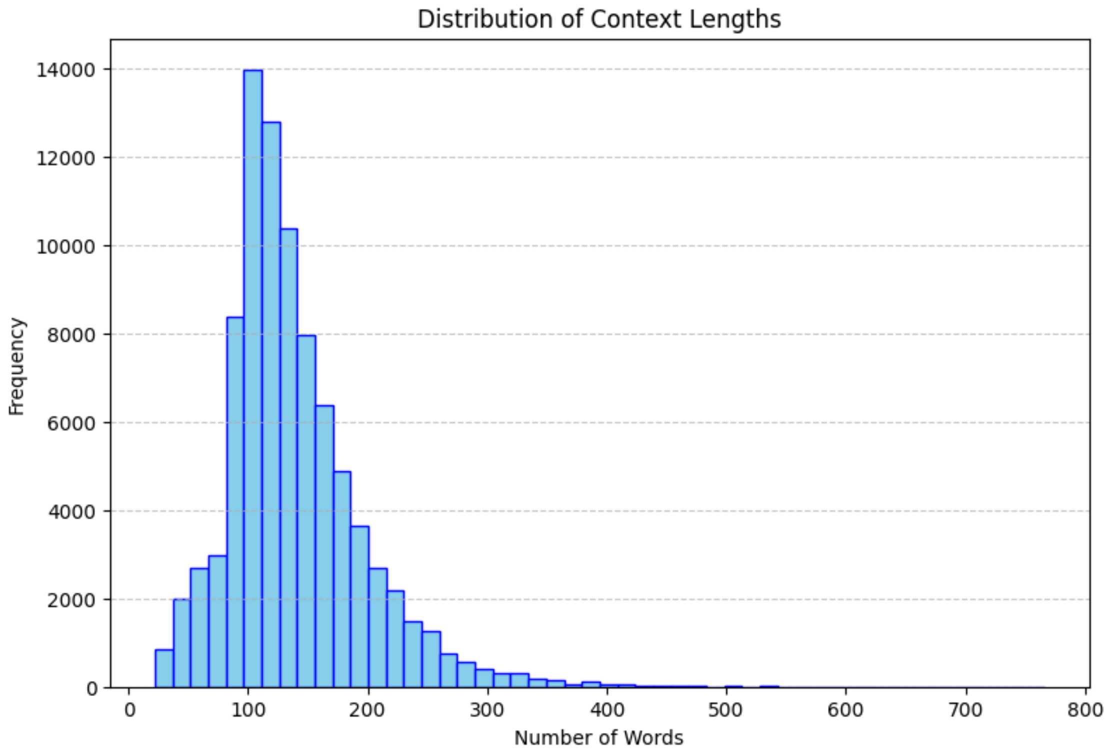
It looks like most of the answers are quite concise, with a majority having less than 10 words.

Distribution of Context Lengths

Now, we will analyze the distribution of the number of words in the contexts. This will give us an understanding of how detailed the contexts in the dataset are.

[ ]:
```
# Tokenize the contexts into words
context_words = [word_tokenize(c) for c in contexts]

# Calculate the length of each context
context_lengths = [len(c) for c in context_words]

# Plot the distribution of context lengths
plt.figure(figsize=(9, 6))
plt.hist(context_lengths, bins=50, color='skyblue', edgecolor='blue')
plt.title('Distribution of Context Lengths')
plt.xlabel('Number of Words')
plt.ylabel('Frequency')
plt.grid(axis='y', linestyle='--', alpha=0.7)
```

```
plt.show()
```



It seems that the contexts have a wide range of lengths, with some being quite short and others being much longer. The avrage centralisig between 100 and 200 words.

Common Words

In this analysis, we'll identify the most common words in the questions, answers, and contexts. This will help us understand the key topics and themes in the dataset.

[ ]:
```
from wordcloud import WordCloud

# Function to generate a word cloud
def generate_word_cloud(words_list, title):
    # Flatten the list of words
    words = [word for sublist in words_list for word in sublist]

    # Generate a word cloud
    wordcloud = WordCloud(width=1000, height=500, background_color='white').
 ↪generate(' '.join(words))

    # Plot the word cloud
    plt.figure(figsize=(10, 5))
    plt.imshow(wordcloud, interpolation='bilinear')
```

```python
    plt.title(title)
    plt.axis('off')
    plt.show()

# Generate word clouds for questions, answers, and contexts
print("Question Word Cloud:")
generate_word_cloud(question_words, 'Word Cloud of Question Text')
print("Answer Word Cloud:")
generate_word_cloud(answer_words, 'Word Cloud of Answer Text')
print("Context Word Cloud:")
generate_word_cloud(context_words, 'Word Cloud of Context Text')
```

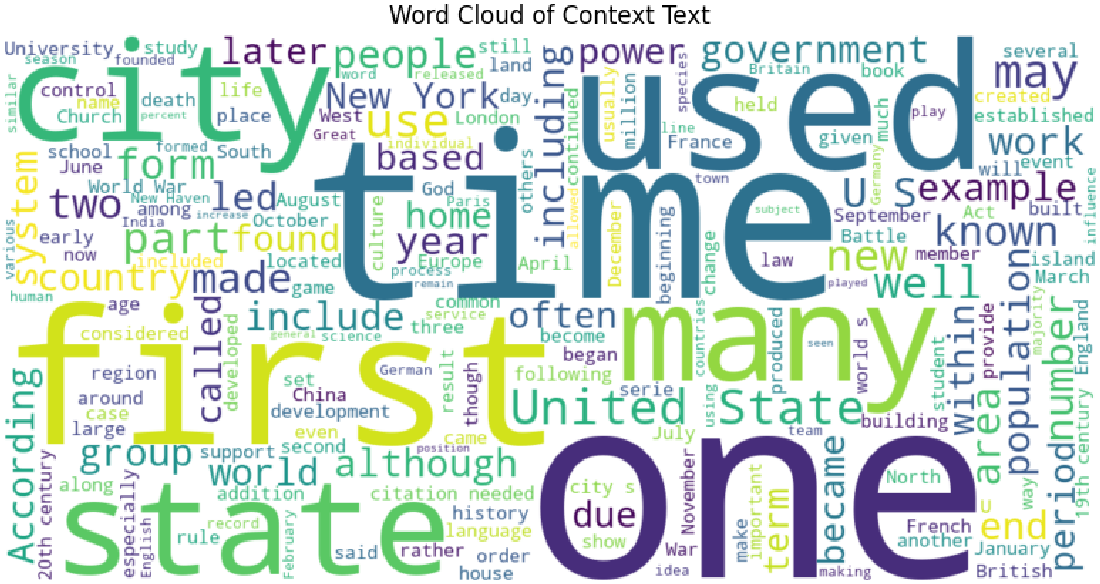
```
Question Word Cloud:
```


```
Answer Word Cloud:
```

Word Cloud of Answer Text


```
Context Word Cloud:
```

Word Cloud of Context Text



The word clouds above provide a visual representation of the most common words in the questions, answers, and contexts. We can see that there are a variety of words in the dataset, including key topics and themes around date/time, states /countries and population.

0.2 Model Selection

In this step, we will select a model architecture for our chatbot.

Given that we are building a question and answer style chatbot, we have selected the BART (Bidirectional and Auto-Regressive Transformers) model. BART is a powerful transformer-based model that is designed for sequence-to-sequence tasks, such as question answering, summarization, and translation.

The BART model has the ability to generate coherent and contextually relevant responses, making it an ideal choice for our chatbot.

We will use the pre-trained BART model provided by the Hugging Face Transformers library, and fine-tune it on our dataset to adapt it to the specific requirements of our chatbot.

0.3 Model Analysis

we will fine-tune the BART model on our dataset and then evaluate its performance. Fine-tuning is the process of training a pre-trained model on a specific task to adapt it to the requirements of that task.

We will proceed witht The following steps:

Fine-Tuning the Model:

Load the pre-trained BART model from the Hugging Face Transformers library.

Fine-tune the model on our dataset.

Evaluating the Model:

Use the fine-tuned model to generate responses to questions.

Evaluate the model's performance using evaluation metrics such as BLEU and ROUGE scores.

[ ]:
```
from transformers import BartTokenizer, BartForConditionalGeneration

# Load the pre-trained BART model and tokenizer
model = BartForConditionalGeneration.from_pretrained('facebook/bart-large')
tokenizer = BartTokenizer.from_pretrained('facebook/bart-large')
```

```
Downloading pytorch_model.bin:   0%|          | 0.00/1.02G [00:00<?, ?B/s]
```

Fine-tuning the model on our dataset involves training the model on our question-answer pairs. We will use the input questions as the input sequences and the corresponding answers as the target sequences. The model will learn to generate the correct answers given the input questions.

[ ]:
```
from torch.nn.utils.rnn import pad_sequence
from torch.utils.data import DataLoader, Dataset
from transformers import AdamW

class QADataset(Dataset):
    def __init__(self, questions, answers, tokenizer, max_length):
        self.questions = questions
```

```
        self.answers = answers
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.questions)

    def __getitem__(self, idx):
        question = self.questions[idx]
        answer = self.answers[idx]

        # Tokenize the question and answer
        inputs = self.tokenizer(question, return_tensors='pt', max_length=self.
 ↪max_length, truncation=True, padding=False)
        target = self.tokenizer(answer, return_tensors='pt', max_length=self.
 ↪max_length, truncation=True, padding=False)

        return {
            'input_ids': inputs['input_ids'].squeeze(),
            'attention_mask': inputs['attention_mask'].squeeze(),
            'labels': target['input_ids'].squeeze(),
        }

def collate_fn(batch):
    input_ids = pad_sequence([item['input_ids'] for item in batch],␣
 ↪batch_first=True, padding_value=1)
    attention_mask = pad_sequence([item['attention_mask'] for item in batch],␣
 ↪batch_first=True, padding_value=0)
    labels = pad_sequence([item['labels'] for item in batch], batch_first=True,␣
 ↪padding_value=-100)

    return {
        'input_ids': input_ids,
        'attention_mask': attention_mask,
        'labels': labels,
    }

# Define the training hyperparameters
batch_size = 16
learning_rate = 5e-5
num_epochs = 3
max_length = 512 # the standard maximum length should be enough condisdering
#                  during our EDA we found the context length to avrage 100 and␣
 ↪200 words.

# Prepare the dataset for training
```

```
dataset = QADataset(questions, answers, tokenizer, max_length)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True,␣
 ↪collate_fn=collate_fn)

# Set up the optimizer and loss function
optimizer = AdamW(model.parameters(), lr=learning_rate)

# Move the model to the GPU
model = model.cuda()

# Set the model to training mode
model.train()

# Training loop
for epoch in range(num_epochs):
    total_loss = 0
    for step , batch in enumerate(dataloader):
        # Move the batch to the GPU
        batch = {k: v.cuda() for k, v in batch.items()}

        # Forward pass
        outputs = model(**batch)
        loss = outputs.loss

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

        # Print the loss every 100 steps
        if (step + 1) % 100 == 0:
            print(f'Epoch {epoch + 1}/{num_epochs}, Step {step + 1}/
 ↪{len(dataloader)}, Loss: {loss.item()}')


    # Calculate the average loss for the epoch
    avg_loss = total_loss / len(dataloader)
    print(f'Epoch {epoch + 1}/{num_epochs}, Loss: {avg_loss}')
```

[ ]:
```
import torch
from transformers import get_linear_schedule_with_warmup
import optuna

# Set the device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```python
# Hyperparameter tuning with Optuna
def objective(trial, model):
    # Define the hyperparameters to be tuned
    batch_size = trial.suggest_categorical('batch_size', [16, 32, 64])
    learning_rate = trial.suggest_loguniform('learning_rate', 1e-6, 1e-4)
    num_epochs = 3

    # Prepare the dataset for training
    dataset = QADataset(questions, answers, tokenizer, max_length)
    dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True, collate_fn=collate_fn)

    # Set up the optimizer and loss function
    optimizer = AdamW(model.parameters(), lr=learning_rate)

    # Set up the learning rate scheduler
    scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=0, num_training_steps=len(dataloader) * num_epochs)

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

    # Set the model to training mode
    model.train()

    # Training loop
    total_loss = 0
    for epoch in range(num_epochs):
        for step, batch in enumerate(dataloader):
            # Move the batch to the GPU
            batch = {k: v.to(device) for k, v in batch.items()}

            # Forward pass
            outputs = model(**batch)
            loss = outputs.loss

            # Backward pass
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            # Update the learning rate
            scheduler.step()

            total_loss += loss.item()
```

```
            # Print the loss every 100 steps
            if (step + 1) % 100 == 0:
                print(f'Epoch {epoch + 1}/{num_epochs}, Step {step + 1}/
↪{len(dataloader)}, Loss: {loss.item()}')

    # Calculate the average loss for all epochs
    avg_loss = total_loss / (len(dataloader) * num_epochs)
    return avg_loss

# Run the hyperparameter tuning
study = optuna.create_study(direction='minimize')
study.optimize(lambda trial: objective(trial, model), n_trials=10)


# Print the best hyperparameters
print('Number of finished trials: ', len(study.trials))
print('Best trial:')
trial = study.best_trial
print('Value: ', trial.value)
print('Params: ')
for key, value in trial.params.items():
    print(f'    {key}: {value}')
```

[ ]:
```
import tensorflow as tf
tf.keras.backend.clear_session()
obj = None
torch.cuda.empty_cache()
```

[ ]:
```
from torch.nn.utils.rnn import pad_sequence
from torch.utils.data import DataLoader, Dataset
from transformers import AdamW
from sklearn.model_selection import train_test_split

class QADataset(Dataset):
    def __init__(self, questions, answers, tokenizer, max_length):
        self.questions = questions
        self.answers = answers
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.questions)

    def __getitem__(self, idx):
        question = self.questions[idx]
        answer = self.answers[idx]
```

```python
        # Tokenize the question and answer
        inputs = self.tokenizer(question, return_tensors='pt', max_length=self.
 ↪max_length, truncation=True, padding=False)
        target = self.tokenizer(answer, return_tensors='pt', max_length=self.
 ↪max_length, truncation=True, padding=False)

        return {
            'input_ids': inputs['input_ids'].squeeze(),
            'attention_mask': inputs['attention_mask'].squeeze(),
            'labels': target['input_ids'].squeeze(),
        }

def collate_fn(batch):
    input_ids = pad_sequence([item['input_ids'] for item in batch],␣
 ↪batch_first=True, padding_value=1)
    attention_mask = pad_sequence([item['attention_mask'] for item in batch],␣
 ↪batch_first=True, padding_value=0)
    labels = pad_sequence([item['labels'] for item in batch], batch_first=True,␣
 ↪padding_value=-100)

    return {
        'input_ids': input_ids,
        'attention_mask': attention_mask,
        'labels': labels,
    }


# Define the training hyperparameters
batch_size = 16  # Optimal value from optimization
learning_rate = 1.593180931067222e-05  # Optimal value from optimization
num_epochs = 3
max_length = 512



# Split the dataset into training and validation sets
questions_train, questions_val, answers_train, answers_val = train_test_split(
    questions, answers, test_size=0.1, random_state=42
)


# Prepare the dataset for training
train_dataset = QADataset(questions_train, answers_train, tokenizer, max_length)
val_dataset = QADataset(questions_val, answers_val, tokenizer, max_length)  #␣
 ↪Validation dataset
```

```python
train_dataloader = DataLoader(train_dataset, batch_size=batch_size,␣
↪shuffle=True, collate_fn=collate_fn)
val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False,␣
↪collate_fn=collate_fn)  # Validation dataloader

# Set up the optimizer
optimizer = AdamW(model.parameters(), lr=learning_rate)

# Move the model to the GPU
model = model.cuda()

# Training loop
for epoch in range(num_epochs):
    model.train()
    total_train_loss = 0
    for step, batch in enumerate(train_dataloader):
        batch = {k: v.cuda() for k, v in batch.items()}
        outputs = model(**batch)
        loss = outputs.loss
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_train_loss += loss.item()
        if (step + 1) % 100 == 0:
            print(f'Epoch {epoch + 1}/{num_epochs}, Step {step + 1}/
↪{len(train_dataloader)}, Loss: {loss.item()}')

    avg_train_loss = total_train_loss / len(train_dataloader)
    print(f'Epoch {epoch + 1}/{num_epochs}, Average Training Loss:␣
↪{avg_train_loss}')

    # Validation loop
    model.eval()
    total_val_loss = 0
    with torch.no_grad():
        for step, batch in enumerate(val_dataloader):
            batch = {k: v.cuda() for k, v in batch.items()}
            outputs = model(**batch)
            loss = outputs.loss
            total_val_loss += loss.item()

    avg_val_loss = total_val_loss / len(val_dataloader)
    print(f'Epoch {epoch + 1}/{num_epochs}, Average Validation Loss:␣
↪{avg_val_loss}')

# Save the fine-tuned model
model.save_pretrained("path/to/save/model")
```

```
tokenizer.save_pretrained("path/to/save/tokenizer")
```

[ ]:
```
import torch
from torch.nn.utils.rnn import pad_sequence
from torch.utils.data import DataLoader, Dataset
from transformers import AdamW
from sklearn.model_selection import train_test_split

class QADataset(Dataset):
    def __init__(self, questions, answers, tokenizer, max_length):
        self.questions = questions
        self.answers = answers
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.questions)

    def __getitem__(self, idx):
        question = self.questions[idx]
        answer = self.answers[idx]

        # Tokenize the question and answer
        inputs = self.tokenizer(question, return_tensors='pt', max_length=self.max_length, truncation=True, padding=False)
        target = self.tokenizer(answer, return_tensors='pt', max_length=self.max_length, truncation=True, padding=False)

        return {
            'input_ids': inputs['input_ids'].squeeze(),
            'attention_mask': inputs['attention_mask'].squeeze(),
            'labels': target['input_ids'].squeeze(),
        }

def collate_fn(batch):
    input_ids = pad_sequence([item['input_ids'] for item in batch], batch_first=True, padding_value=1)
    attention_mask = pad_sequence([item['attention_mask'] for item in batch], batch_first=True, padding_value=0)
    labels = pad_sequence([item['labels'] for item in batch], batch_first=True, padding_value=-100)

    return {
        'input_ids': input_ids,
        'attention_mask': attention_mask,
        'labels': labels,
    }
```

```python
# Define the training hyperparameters
batch_size = 16  # Optimal value from optimization
learning_rate = 1.593180931067222e-05  # Optimal value from optimization
num_epochs = 3
max_length = 512



# Split the dataset into training and validation sets
questions_train, questions_val, answers_train, answers_val = train_test_split(
    questions, answers, test_size=0.1, random_state=42
)


# Prepare the dataset for training
train_dataset = QADataset(questions_train, answers_train, tokenizer, max_length)
val_dataset = QADataset(questions_val, answers_val, tokenizer, max_length)  # Validation dataset

train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, collate_fn=collate_fn)
val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False, collate_fn=collate_fn)  # Validation dataloader

# Set up the optimizer
optimizer = AdamW(model.parameters(), lr=learning_rate)

# Move the model to the GPU
model = model.cuda()

# Early stopping parameters
patience = 2
best_val_loss = float('inf')
patience_counter = 0

# Training loop with early stopping
for epoch in range(num_epochs):
    total_loss = 0
    model.train()
    for step, batch in enumerate(train_dataloader):
        # Move the batch to the GPU
        batch = {k: v.cuda() for k, v in batch.items()}

        # Forward pass
        outputs = model(**batch)
```

```
        loss = outputs.loss

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

        # Print the loss every 100 steps
        if (step + 1) % 100 == 0:
            print(f'Epoch {epoch + 1}/{num_epochs}, Step {step + 1}/
↪{len(train_dataloader)}, Loss: {loss.item()}')

    # Calculate the average training loss for the epoch
    avg_train_loss = total_loss / len(train_dataloader)
    print(f'Epoch {epoch + 1}/{num_epochs}, Average Training Loss:␣
↪{avg_train_loss}')

    # Evaluate the model on the validation set
    model.eval()
    total_val_loss = 0
    with torch.no_grad():
        for batch in val_dataloader:
            # Move the batch to the GPU
            batch = {k: v.cuda() for k, v in batch.items()}

            # Forward pass
            outputs = model(**batch)
            loss = outputs.loss

            total_val_loss += loss.item()

    # Calculate the average validation loss for the epoch
    avg_val_loss = total_val_loss / len(val_dataloader)
    print(f'Epoch {epoch + 1}/{num_epochs}, Average Validation Loss:␣
↪{avg_val_loss}')

    # Check for early stopping
    if avg_val_loss < best_val_loss:
        best_val_loss = avg_val_loss
        patience_counter = 0
    else:
        patience_counter += 1
        if patience_counter >= patience:
            print('Early stopping!')
            break
```

[ ]:
```python
import torch
from torch.nn.utils.rnn import pad_sequence
from torch.utils.data import DataLoader, Dataset
from transformers import AdamW, get_linear_schedule_with_warmup
from sklearn.model_selection import train_test_split
from torch.cuda.amp import autocast, GradScaler

class QADataset(Dataset):
    def __init__(self, questions, answers, tokenizer, max_length):
        self.questions = questions
        self.answers = answers
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.questions)

    def __getitem__(self, idx):
        question = self.questions[idx]
        answer = self.answers[idx]

        # Tokenize the question and answer
        inputs = self.tokenizer(question, return_tensors='pt', max_length=self.
↪max_length, truncation=True, padding=False)
        target = self.tokenizer(answer, return_tensors='pt', max_length=self.
↪max_length, truncation=True, padding=False)

        return {
            'input_ids': inputs['input_ids'].squeeze(),
            'attention_mask': inputs['attention_mask'].squeeze(),
            'labels': target['input_ids'].squeeze(),
        }

def collate_fn(batch):
    input_ids = pad_sequence([item['input_ids'] for item in batch],␣
↪batch_first=True, padding_value=1)
    attention_mask = pad_sequence([item['attention_mask'] for item in batch],␣
↪batch_first=True, padding_value=0)
    labels = pad_sequence([item['labels'] for item in batch], batch_first=True,␣
↪padding_value=-100)

    return {
        'input_ids': input_ids,
        'attention_mask': attention_mask,
        'labels': labels,
    }
```

```
# Define the training hyperparameters
batch_size = 16
learning_rate = 1.593180931067222e-05
num_epochs = 3
max_length = 512

# Split the dataset into training and validation sets
questions_train, questions_val, answers_train, answers_val = train_test_split(
    questions, answers, test_size=0.1, random_state=42
)

# Prepare the dataset for training
train_dataset = QADataset(questions_train, answers_train, tokenizer, max_length)
val_dataset = QADataset(questions_val, answers_val, tokenizer, max_length)

train_dataloader = DataLoader(train_dataset, batch_size=batch_size,␣
↪shuffle=True, collate_fn=collate_fn)
val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False,␣
↪collate_fn=collate_fn)

# Set up the optimizer with weight decay
optimizer = AdamW(model.parameters(), lr=learning_rate, weight_decay=0.01)

# Move the model to the GPU
model = model.cuda()

# Early stopping parameters
patience = 2
best_val_loss = float('inf')
patience_counter = 0

# Set up the learning rate scheduler with warm-up steps
num_training_steps = len(train_dataloader) * num_epochs
num_warmup_steps = int(0.1 * num_training_steps)  # 10% of training steps for␣
↪warm-up
scheduler = get_linear_schedule_with_warmup(optimizer,␣
↪num_warmup_steps=num_warmup_steps, num_training_steps=num_training_steps)

# Mixed precision training
scaler = GradScaler()

# Training loop with early stopping
for epoch in range(num_epochs):
    total_loss = 0
    model.train()
    for step, batch in enumerate(train_dataloader):
```

```python
        # Move the batch to the GPU
        batch = {k: v.cuda() for k, v in batch.items()}

        # Forward pass with mixed precision
        with autocast():
            outputs = model(**batch)
            loss = outputs.loss

        # Backward pass with gradient clipping and mixed precision
        optimizer.zero_grad()
        scaler.scale(loss).backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        scaler.step(optimizer)
        scaler.update()

        total_loss += loss.item()

        # Print the loss every 100 steps
        if (step + 1) % 100 == 0:
            print(f'Epoch {epoch + 1}/{num_epochs}, Step {step + 1}/
↪{len(train_dataloader)}, Loss: {loss.item()}')

    # Update the learning rate
    scheduler.step()

    # Calculate the average training loss for the epoch
    avg_train_loss = total_loss / len(train_dataloader)
    print(f'Epoch {epoch + 1}/{num_epochs}, Average Training Loss:␣
↪{avg_train_loss}')

    # Evaluate the model on the validation set
    model.eval()
    total_val_loss = 0
    with torch.no_grad():
        for batch in val_dataloader:
            # Move the batch to the GPU
            batch = {k: v.cuda() for k, v in batch.items()}

            # Forward pass
            outputs = model(**batch)
            loss = outputs.loss

            total_val_loss += loss.item()

    # Calculate the average validation loss for the epoch
    avg_val_loss = total_val_loss / len(val_dataloader)
```

```
    print(f'Epoch {epoch + 1}/{num_epochs}, Average Validation Loss:␣
↪{avg_val_loss}')

    # Check for early stopping
    if avg_val_loss < best_val_loss:
        best_val_loss = avg_val_loss
        patience_counter = 0
    else:
        patience_counter += 1
        if patience_counter >= patience:
            print('Early stopping!')
            break
```

[ ]:
```
import torch
from torch.nn.utils.rnn import pad_sequence
from torch.utils.data import DataLoader, Dataset
from transformers import AdamW, get_linear_schedule_with_warmup
from sklearn.model_selection import train_test_split
from torch.cuda.amp import autocast, GradScaler

class QADataset(Dataset):
    def __init__(self, questions, answers, tokenizer, max_length):
        self.questions = questions
        self.answers = answers
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.questions)

    def __getitem__(self, idx):
        question = self.questions[idx]
        answer = self.answers[idx]

        # Tokenize the question and answer
        inputs = self.tokenizer(question, return_tensors='pt', max_length=self.
↪max_length, truncation=True, padding=False)
        target = self.tokenizer(answer, return_tensors='pt', max_length=self.
↪max_length, truncation=True, padding=False)

        return {
            'input_ids': inputs['input_ids'].squeeze(),
            'attention_mask': inputs['attention_mask'].squeeze(),
            'labels': target['input_ids'].squeeze(),
        }

def collate_fn(batch):
```

```
    input_ids = pad_sequence([item['input_ids'] for item in batch],␣
↪batch_first=True, padding_value=1)
    attention_mask = pad_sequence([item['attention_mask'] for item in batch],␣
↪batch_first=True, padding_value=0)
    labels = pad_sequence([item['labels'] for item in batch], batch_first=True,␣
↪padding_value=-100)

    return {
        'input_ids': input_ids,
        'attention_mask': attention_mask,
        'labels': labels,
    }


# Define the training hyperparameters
batch_size = 16
learning_rate = 1.593180931067222e-05
num_epochs = 3
max_length = 512

# Split the dataset into training and validation sets
questions_train, questions_val, answers_train, answers_val = train_test_split(
    questions, answers, test_size=0.1, random_state=42
)

# Prepare the dataset for training
train_dataset = QADataset(questions_train, answers_train, tokenizer, max_length)
val_dataset = QADataset(questions_val, answers_val, tokenizer, max_length)

train_dataloader = DataLoader(train_dataset, batch_size=batch_size,␣
↪shuffle=True, collate_fn=collate_fn)
val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False,␣
↪collate_fn=collate_fn)

# Set up the optimizer with weight decay
optimizer = AdamW(model.parameters(), lr=learning_rate, weight_decay=0.01)

# Move the model to the GPU
model = model.cuda()

# Early stopping parameters
patience = 2
best_val_loss = float('inf')
patience_counter = 0

# Set up the learning rate scheduler with warm-up steps
num_training_steps = len(train_dataloader) * num_epochs
```

```
num_warmup_steps = int(0.1 * num_training_steps)  # 10% of training steps for␣
↪warm-up
scheduler = get_linear_schedule_with_warmup(optimizer,␣
↪num_warmup_steps=num_warmup_steps, num_training_steps=num_training_steps)

# Mixed precision training
scaler = GradScaler()

# Training loop with early stopping
for epoch in range(num_epochs):
    total_loss = 0
    model.train()
    for step, batch in enumerate(train_dataloader):
        # Move the batch to the GPU
        batch = {k: v.cuda() for k, v in batch.items()}

        # Forward pass with mixed precision
        with autocast():
            outputs = model(**batch)
            loss = outputs.loss

        # Backward pass with gradient clipping and mixed precision
        optimizer.zero_grad()
        scaler.scale(loss).backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        scaler.step(optimizer)
        scaler.update()

        total_loss += loss.item()

        # Print the loss every 100 steps
        if (step + 1) % 100 == 0:
            print(f'Epoch {epoch + 1}/{num_epochs}, Step {step + 1}/
↪{len(train_dataloader)}, Loss: {loss.item()}')

    # Update the learning rate
    scheduler.step()

    # Calculate the average training loss for the epoch
    avg_train_loss = total_loss / len(train_dataloader)
    print(f'Epoch {epoch + 1}/{num_epochs}, Average Training Loss:␣
↪{avg_train_loss}')

    # Evaluate the model on the validation set
    model.eval()
    total_val_loss = 0
    with torch.no_grad():
```

```
        for batch in val_dataloader:
            # Move the batch to the GPU
            batch = {k: v.cuda() for k, v in batch.items()}

            # Forward pass
            outputs = model(**batch)
            loss = outputs.loss

            total_val_loss += loss.item()

    # Calculate the average validation loss for the epoch
    avg_val_loss = total_val_loss / len(val_dataloader)
    print(f'Epoch {epoch + 1}/{num_epochs}, Average Validation Loss:␣
 ↪{avg_val_loss}')

    # Check for early stopping
    if avg_val_loss < best_val_loss:
        best_val_loss = avg_val_loss
        patience_counter = 0
    else:
        patience_counter += 1
        if patience_counter >= patience:
            print('Early stopping!')
            break
```

[ ]:

```
import torch
from torch.nn.utils.rnn import pad_sequence
from torch.utils.data import DataLoader, Dataset
from transformers import AdamW, get_linear_schedule_with_warmup
from sklearn.model_selection import train_test_split
from torch.cuda.amp import autocast, GradScaler
from sklearn.metrics import f1_score

class QADataset(Dataset):
    def __init__(self, questions, answers, tokenizer, max_length):
        self.questions = questions
        self.answers = answers
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.questions)

    def __getitem__(self, idx):
        question = self.questions[idx]
        answer = self.answers[idx]
```

```
        # Tokenize the question and answer
        inputs = self.tokenizer(question, return_tensors='pt', max_length=self.
↪max_length, truncation=True, padding=False)
        target = self.tokenizer(answer, return_tensors='pt', max_length=self.
↪max_length, truncation=True, padding=False)

        return {
            'input_ids': inputs['input_ids'].squeeze(),
            'attention_mask': inputs['attention_mask'].squeeze(),
            'labels': target['input_ids'].squeeze(),
        }

def collate_fn(batch):
    input_ids = pad_sequence([item['input_ids'] for item in batch],␣
↪batch_first=True, padding_value=1)
    attention_mask = pad_sequence([item['attention_mask'] for item in batch],␣
↪batch_first=True, padding_value=0)
    labels = pad_sequence([item['labels'] for item in batch], batch_first=True,␣
↪padding_value=-100)

    return {
        'input_ids': input_ids,
        'attention_mask': attention_mask,
        'labels': labels,
    }

# Define the training hyperparameters
batch_size = 16
learning_rate = 1.593180931067222e-05
num_epochs = 3
max_length = 512

# Split the dataset into training and validation sets
questions_train, questions_val, answers_train, answers_val = train_test_split(
    questions, answers, test_size=0.1, random_state=42
)

# Prepare the dataset for training
train_dataset = QADataset(questions_train, answers_train, tokenizer, max_length)
val_dataset = QADataset(questions_val, answers_val, tokenizer, max_length)

train_dataloader = DataLoader(train_dataset, batch_size=batch_size,␣
↪shuffle=True, collate_fn=collate_fn)
val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False,␣
↪collate_fn=collate_fn)
```

```
# Set up the optimizer with weight decay
optimizer = AdamW(model.parameters(), lr=learning_rate, weight_decay=0.01)

# Move the model to the GPU
model = model.cuda()

# Early stopping parameters
patience = 2
best_val_loss = float('inf')
patience_counter = 0

# Set up the learning rate scheduler with warm-up steps
num_training_steps = len(train_dataloader) * num_epochs
num_warmup_steps = int(0.1 * num_training_steps)  # 10% of training steps for␣
↪warm-up
scheduler = get_linear_schedule_with_warmup(optimizer,␣
↪num_warmup_steps=num_warmup_steps, num_training_steps=num_training_steps)

# Mixed precision training
scaler = GradScaler()

# Gradual Unfreezing: Freeze all layers except the last one initially
for param in model.parameters():
    param.requires_grad = False
for param in model.model.decoder.layers[-1].parameters():
    param.requires_grad = True

# Training loop with early stopping
for epoch in range(num_epochs):
    total_loss = 0
    model.train()
    for step, batch in enumerate(train_dataloader):
        # Move the batch to the GPU
        batch = {k: v.cuda() for k, v in batch.items()}

        # Forward pass with mixed precision
        with autocast():
            outputs = model(**batch)
            loss = outputs.loss

        # Backward pass with gradient clipping and mixed precision
        optimizer.zero_grad()
        scaler.scale(loss).backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        scaler.step(optimizer)
        scaler.update()
```

```
        total_loss += loss.item()

        # Print the loss every 100 steps
        if (step + 1) % 100 == 0:
            print(f'Epoch {epoch + 1}/{num_epochs}, Step {step + 1}/
↪{len(train_dataloader)}, Loss: {loss.item()}')

    # Update the learning rate
    scheduler.step()

    # Calculate the average training loss for the epoch
    avg_train_loss = total_loss / len(train_dataloader)
    print(f'Epoch {epoch + 1}/{num_epochs}, Average Training Loss:␣
↪{avg_train_loss}')

    # Evaluate the model on the validation set
    model.eval()
    total_val_loss = 0
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for batch in val_dataloader:
            # Move the batch to the GPU
            batch = {k: v.cuda() for k, v in batch.items()}

            # Forward pass
            outputs = model(**batch)
            loss = outputs.loss

            total_val_loss += loss.item()

            # Calculate F1 Score
            logits = outputs.logits
            preds = torch.argmax(logits, dim=-1)
            labels = batch['labels']

            all_preds.extend(preds.cpu().numpy().flatten())
            all_labels.extend(labels.cpu().numpy().flatten())

    # Calculate the average validation loss for the epoch
    avg_val_loss = total_val_loss / len(val_dataloader)
    print(f'Epoch {epoch + 1}/{num_epochs}, Average Validation Loss:␣
↪{avg_val_loss}')

    # Calculate F1 Score
    f1 = f1_score(all_labels, all_preds, average='macro')
    print(f'Epoch {epoch + 1}/{num_epochs}, F1 Score: {f1}')
```

```
    # Check for early stopping
    if avg_val_loss < best_val_loss:
        best_val_loss = avg_val_loss
        patience_counter = 0
    else:
        patience_counter += 1
        if patience_counter >= patience:
            print('Early stopping!')
            break

    # Gradual Unfreezing: Unfreeze one more layer
    if epoch < num_epochs - 1:
        for param in model.model.decoder.layers[-(epoch + 2)].parameters():
            param.requires_grad = True
```

[ ]:

```
import torch
from torch.nn.utils.rnn import pad_sequence
from torch.utils.data import DataLoader, Dataset
from transformers import AdamW, get_linear_schedule_with_warmup
from sklearn.model_selection import train_test_split
from torch.cuda.amp import autocast, GradScaler
from sklearn.metrics import f1_score

class QADataset(Dataset):
    def __init__(self, questions, answers, tokenizer, max_length):
        self.questions = questions
        self.answers = answers
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.questions)

    def __getitem__(self, idx):
        question = self.questions[idx]
        answer = self.answers[idx]

        # Tokenize the question and answer
        inputs = self.tokenizer(question, return_tensors='pt', max_length=self.
↪max_length, truncation=True, padding=False)
        target = self.tokenizer(answer, return_tensors='pt', max_length=self.
↪max_length, truncation=True, padding=False)

        return {
            'input_ids': inputs['input_ids'].squeeze(),
            'attention_mask': inputs['attention_mask'].squeeze(),
```

```
            'labels': target['input_ids'].squeeze(),
        }

def collate_fn(batch):
    input_ids = pad_sequence([item['input_ids'] for item in batch], batch_first=True, padding_value=1)
    attention_mask = pad_sequence([item['attention_mask'] for item in batch], batch_first=True, padding_value=0)
    labels = pad_sequence([item['labels'] for item in batch], batch_first=True, padding_value=-100)

    return {
        'input_ids': input_ids,
        'attention_mask': attention_mask,
        'labels': labels,
    }

# Define the training hyperparameters
batch_size = 16
learning_rate = 1.593180931067222e-05
num_epochs = 3
max_length = 512

# Split the dataset into training and validation sets
questions_train, questions_val, answers_train, answers_val = train_test_split(
    questions, answers, test_size=0.1, random_state=42
)

# Prepare the dataset for training
train_dataset = QADataset(questions_train, answers_train, tokenizer, max_length)
val_dataset = QADataset(questions_val, answers_val, tokenizer, max_length)

train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, collate_fn=collate_fn)
val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False, collate_fn=collate_fn)

# Set up the optimizer with weight decay
optimizer = AdamW(model.parameters(), lr=learning_rate, weight_decay=0.01)

# Move the model to the GPU
model = model.cuda()

# Early stopping parameters
patience = 2
best_val_loss = float('inf')
patience_counter = 0
```

```python
# Set up the learning rate scheduler with warm-up steps
num_training_steps = len(train_dataloader) * num_epochs
num_warmup_steps = int(0.1 * num_training_steps)  # 10% of training steps for warm-up
scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=num_warmup_steps, num_training_steps=num_training_steps)

# Mixed precision training
scaler = GradScaler()

# Gradual Unfreezing: Freeze all layers except the last one initially
for param in model.parameters():
    param.requires_grad = False
for param in model.model.decoder.layers[-1].parameters():
    param.requires_grad = True

# Training loop with early stopping
for epoch in range(num_epochs):
    total_loss = 0
    model.train()
    for step, batch in enumerate(train_dataloader):
        # Move the batch to the GPU
        batch = {k: v.cuda() for k, v in batch.items()}

        # Forward pass with mixed precision
        with autocast():
            outputs = model(**batch)
            loss = outputs.loss

        # Backward pass with gradient clipping and mixed precision
        optimizer.zero_grad()
        scaler.scale(loss).backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        scaler.step(optimizer)
        scaler.update()

        total_loss += loss.item()

        # Print the loss every 100 steps
        if (step + 1) % 100 == 0:
            print(f'Epoch {epoch + 1}/{num_epochs}, Step {step + 1}/{len(train_dataloader)}, Loss: {loss.item()}')

    # Update the learning rate
    scheduler.step()
```

```python
    # Calculate the average training loss for the epoch
    avg_train_loss = total_loss / len(train_dataloader)
    print(f'Epoch {epoch + 1}/{num_epochs}, Average Training Loss:␣
↪{avg_train_loss}')

    # Evaluate the model on the validation set
    model.eval()
    total_val_loss = 0
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for batch in val_dataloader:
            # Move the batch to the GPU
            batch = {k: v.cuda() for k, v in batch.items()}

            # Forward pass
            outputs = model(**batch)
            loss = outputs.loss

            total_val_loss += loss.item()

            # Calculate F1 Score
            logits = outputs.logits
            preds = torch.argmax(logits, dim=-1)
            labels = batch['labels']

            all_preds.extend(preds.cpu().numpy().flatten())
            all_labels.extend(labels.cpu().numpy().flatten())

    # Calculate the average validation loss for the epoch
    avg_val_loss = total_val_loss / len(val_dataloader)
    print(f'Epoch {epoch + 1}/{num_epochs}, Average Validation Loss:␣
↪{avg_val_loss}')

    # Calculate F1 Score
    f1 = f1_score(all_labels, all_preds, average='macro')
    print(f'Epoch {epoch + 1}/{num_epochs}, F1 Score: {f1}')

    # Check for early stopping
    if avg_val_loss < best_val_loss:
        best_val_loss = avg_val_loss
        patience_counter = 0
    else:
        patience_counter += 1
        if patience_counter >= patience:
            print('Early stopping!')
            break
```

```python
    # Gradual Unfreezing: Unfreeze one more layer
    if epoch < num_epochs - 1:
        for param in model.model.decoder.layers[-(epoch + 2)].parameters():
            param.requires_grad = True
```

[ ]:
```python
import torch
from torch.nn.utils.rnn import pad_sequence
from torch.utils.data import DataLoader, Dataset
from transformers import AdamW, get_linear_schedule_with_warmup
from sklearn.model_selection import train_test_split
from torch.cuda.amp import autocast, GradScaler
from sklearn.metrics import f1_score
from nltk.translate.bleu_score import corpus_bleu
import numpy as np


class QADataset(Dataset):
    def __init__(self, questions, answers, tokenizer, max_length):
        self.questions = questions
        self.answers = answers
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.questions)

    def __getitem__(self, idx):
        question = self.questions[idx]
        answer = self.answers[idx]

        # Tokenize the question and answer
        inputs = self.tokenizer(question, return_tensors='pt', max_length=self.
 ↪max_length, truncation=True, padding=False)
        target = self.tokenizer(answer, return_tensors='pt', max_length=self.
 ↪max_length, truncation=True, padding=False)

        return {
            'input_ids': inputs['input_ids'].squeeze(),
            'attention_mask': inputs['attention_mask'].squeeze(),
            'labels': target['input_ids'].squeeze(),
        }

def collate_fn(batch):
    input_ids = pad_sequence([item['input_ids'] for item in batch],␣
 ↪batch_first=True, padding_value=1)
```

```
    attention_mask = pad_sequence([item['attention_mask'] for item in batch],␣
↪batch_first=True, padding_value=0)
    labels = pad_sequence([item['labels'] for item in batch], batch_first=True,␣
↪padding_value=-100)

    return {
        'input_ids': input_ids,
        'attention_mask': attention_mask,
        'labels': labels,
    }

# Hyperparameter tuning: experiment with different learning rates and batch␣
↪sizes
learning_rates = [1.593180931067222e-05, 1e-5, 1e-6]
batch_sizes = [16, 32, 64]

for lr in learning_rates:
    for bs in batch_sizes:
        # Update the batch size and learning rate in your code
        batch_size = bs
        learning_rate = lr

        # Rest of the training code

        # Define the training hyperparameters
        num_epochs = 3
        max_length = 512

        # Split the dataset into training and validation sets
        questions_train, questions_val, answers_train, answers_val =␣
↪train_test_split(
            questions, answers, test_size=0.1, random_state=42
        )

        # Prepare the dataset for training
        train_dataset = QADataset(questions_train, answers_train, tokenizer,␣
↪max_length)
        val_dataset = QADataset(questions_val, answers_val, tokenizer,␣
↪max_length)

        train_dataloader = DataLoader(train_dataset, batch_size=batch_size,␣
↪shuffle=True, collate_fn=collate_fn)
        val_dataloader = DataLoader(val_dataset, batch_size=batch_size,␣
↪shuffle=False, collate_fn=collate_fn)

        # Set up the optimizer with weight decay
```

```
    optimizer = AdamW(model.parameters(), lr=learning_rate, weight_decay=0.
↪01)

    # Move the model to the GPU
    model = model.cuda()

    # Early stopping parameters
    patience = 2
    best_val_loss = float('inf')
    patience_counter = 0

    # Set up the learning rate scheduler with warm-up steps
    num_training_steps = len(train_dataloader) * num_epochs
    num_warmup_steps = int(0.1 * num_training_steps)  # 10% of training␣
↪steps for warm-up
    scheduler = get_linear_schedule_with_warmup(optimizer,␣
↪num_warmup_steps=num_warmup_steps, num_training_steps=num_training_steps)

    # Mixed precision training
    scaler = GradScaler()

    # Gradual Unfreezing: Freeze all layers except the last one initially
    for param in model.parameters():
        param.requires_grad = False
    for param in model.model.decoder.layers[-1].parameters():
        param.requires_grad = True

    # Training loop with early stopping
    for epoch in range(num_epochs):
        total_loss = 0
        model.train()
        for step, batch in enumerate(train_dataloader):
            # Move the batch to the GPU
            batch = {k: v.cuda() for k, v in batch.items()}

            # Forward pass with mixed precision
            with autocast():
                outputs = model(**batch)
                loss = outputs.loss

            # Backward pass with gradient clipping and mixed precision
            optimizer.zero_grad()
            scaler.scale(loss).backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
            scaler.step(optimizer)
            scaler.update()
```

```python
            total_loss += loss.item()

            # Print the loss every 100 steps
            if (step + 1) % 100 == 0:
                print(f'Epoch {epoch + 1}/{num_epochs}, Step {step + 1}/{len(train_dataloader)}, Loss: {loss.item()}')

        # Update the learning rate
        scheduler.step()

        # Calculate the average training loss for the epoch
        avg_train_loss = total_loss / len(train_dataloader)
        print(f'Epoch {epoch + 1}/{num_epochs}, Average Training Loss: {avg_train_loss}')

        # Evaluate the model on the validation set
        model.eval()
        total_val_loss = 0
        all_preds = []
        all_labels = []
        with torch.no_grad():
            for batch in val_dataloader:
                # Move the batch to the GPU
                batch = {k: v.cuda() for k, v in batch.items()}

                # Forward pass
                outputs = model(**batch)
                loss = outputs.loss

                total_val_loss += loss.item()

                # Calculate F1 Score
                logits = outputs.logits
                preds = torch.argmax(logits, dim=-1)
                labels = batch['labels']

                all_preds.extend(preds.cpu().numpy().flatten())
                all_labels.extend(labels.cpu().numpy().flatten())

        # Calculate the average validation loss for the epoch
        avg_val_loss = total_val_loss / len(val_dataloader)
        print(f'Epoch {epoch + 1}/{num_epochs}, Average Validation Loss: {avg_val_loss}')

        # Calculate F1 Score
        f1 = f1_score(all_labels, all_preds, average='macro')
        print(f'Epoch {epoch + 1}/{num_epochs}, F1 Score: {f1}')
```

```
            # Calculate BLEU Score

           # Filter out invalid values from all_labels
           # Filter out invalid values from all_labels
           # Filter out invalid values from all_labels
            valid_labels = []
            for labels in all_labels:
                if labels is not None:
                    if isinstance(labels, np.ndarray):
                        labels_list = labels.tolist()
                    elif isinstance(labels, np.int64):
                        labels_list = [int(labels)]
                    else:
                        labels_list = labels
                    if all(isinstance(label, int) for label in labels_list):
                        valid_labels.append(labels_list)

            # Flatten the valid_labels list
            flat_valid_labels = [item for sublist in valid_labels for item in␣
↪sublist]




            pred_sentences = tokenizer.batch_decode(all_preds,␣
↪skip_special_tokens=True)

            # Decode the valid labels
            label_sentences = tokenizer.batch_decode(flat_valid_labels,␣
↪skip_special_tokens=True)

            bleu_score = corpus_bleu([[label.split()] for label in␣
↪label_sentences], [pred.split() for pred in pred_sentences])

            print(f'Epoch {epoch + 1}/{num_epochs}, BLEU Score: {bleu_score}')

            # Check for early stopping
            if avg_val_loss < best_val_loss:
                best_val_loss = avg_val_loss
                patience_counter = 0
            else:
                patience_counter += 1
                if patience_counter >= patience:
                    print('Early stopping!')
                    break
```

```
            # Gradual Unfreezing: Unfreeze one more layer
            if epoch < num_epochs - 1:
                for param in model.model.decoder.layers[-(epoch + 2)].
↪parameters():
                    param.requires_grad = True
```

[ ]:
```
from nltk.translate.bleu_score import sentence_bleu, SmoothingFunction

# Evaluate the model on the validation set
model.eval()
all_preds = []
all_labels = []
with torch.no_grad():
    for batch in val_dataloader:
        # Move the batch to the GPU
        batch = {k: v.cuda() for k, v in batch.items()}

        # Forward pass
        outputs = model(**batch)

        # Decode the predicted answers
        logits = outputs.logits
        preds = torch.argmax(logits, dim=-1)
        pred_ans = tokenizer.batch_decode(preds, skip_special_tokens=True)

        # Decode the actual answers
        labels = batch['labels']
        labels = labels[labels != -100]  # Remove -100 values
        actual_ans = tokenizer.batch_decode(labels, skip_special_tokens=True)

        all_preds.extend(pred_ans)
        all_labels.extend(actual_ans)

# Calculate BLEU score
bleu_score = 0
for pred, label in zip(all_preds, all_labels):
    reference = [label.split()]
    candidate = pred.split()
    bleu_score += sentence_bleu(reference, candidate,␣
↪smoothing_function=SmoothingFunction().method1)

avg_bleu_score = bleu_score / len(all_preds)
print(f'Average BLEU Score: {avg_bleu_score}')
```

```
Average BLEU Score: 0.00014017637825658317
```

BERT Modeling notebook

October 23, 2023

[ ]:
```python
# LangChain
from langchain.document_loaders import UnstructuredFileLoader
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import Pinecone

# NLTK
import nltk
nltk.download('punkt')
from nltk.tokenize import sent_tokenize
from langchain.text_splitter import NLTKTextSplitter

import openai

# Pinecone
import pinecone

import os
import streamlit as st
import requests
from dotenv import load_dotenv
import json
import pandas as pd
import numpy as np

load_dotenv()
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     /Users/trevormcgirr/nltk_data…
[nltk_data]   Package punkt is already up-to-date!
```

[ ]: True

[ ]:
```python
# PINECONE_API_KEY = os.environ.get("PINECONE_API_KEY")
# PINECONE_INDEX_NAME = os.environ.get("PINECONE_INDEX_NAME")
PINECONE_API_KEY = ""
PINECONE_INDEX_NAME = "gcp-starter"

OPENAI_API_KEY = os.environ.get("OPENAI_API_KEY")
```

```
openai.api_key = OPENAI_API_KEY

pinecone.init(
 api_key=PINECONE_API_KEY,
 environment=PINECONE_INDEX_NAME
)
index = pinecone.Index('chatbot')

# OpenAI Embeddings
embeddings = OpenAIEmbeddings()
```

[ ]:
```
# https://www.kaggle.com/code/sanjay11100/
 ↪squad-stanford-q-a-json-to-pandas-dataframe
def squad_json_to_dataframe_train(input_file_path, record_path =␣
 ↪['data','paragraphs','qas','answers'],
                           verbose = 1):
    """
    input_file_path: path to the squad json file.
    record_path: path to deepest level in json file default value is
    ['data','paragraphs','qas','answers']
    verbose: 0 to suppress it default is 1
    """
    if verbose:
        print("Reading the json file")
    file = json.loads(open(input_file_path).read())
    if verbose:
        print("processing...")
    # parsing different level's in the json file
    js = pd.io.json.json_normalize(file , record_path )
    m = pd.io.json.json_normalize(file, record_path[:-1] )
    r = pd.io.json.json_normalize(file,record_path[:-2])

    #combining it into single dataframe
    idx = np.repeat(r['context'].values, r.qas.str.len())
    ndx  = np.repeat(m['id'].values,m['answers'].str.len())
    m['context'] = idx
    js['q_idx'] = ndx
    main = pd.concat([ m[['id','question','context']].set_index('id'),js.
 ↪set_index('q_idx')],1,sort=False).reset_index()
    main['c_id'] = main['context'].factorize()[0]
    if verbose:
        print("shape of the dataframe is {}".format(main.shape))
        print("Done")
    return main

def squad_json_to_dataframe_dev(input_file_path, record_path =␣
 ↪['data','paragraphs','qas','answers'],
```

```
                                verbose = 1):
    """
    input_file_path: path to the squad json file.
    record_path: path to deepest level in json file default value is
    ['data','paragraphs','qas','answers']
    verbose: 0 to suppress it default is 1
    """
    if verbose:
        print("Reading the json file")
    file = json.loads(open(input_file_path).read())
    if verbose:
        print("processing...")
    # parsing different level's in the json file
    js = pd.io.json.json_normalize(file , record_path )
    m = pd.io.json.json_normalize(file, record_path[:-1] )
    r = pd.io.json.json_normalize(file,record_path[:-2])

    #combining it into single dataframe
    idx = np.repeat(r['context'].values, r.qas.str.len())
#     ndx  = np.repeat(m['id'].values,m['answers'].str.len())
    m['context'] = idx
#     js['q_idx'] = ndx
    main = m[['id','question','context','answers']].set_index('id').
 ↪reset_index()
    main['c_id'] = main['context'].factorize()[0]
    if verbose:
        print("shape of the dataframe is {}".format(main.shape))
        print("Done")
    return main
```

[ ]:
```
# training data
# input_file_path = '../input/train-v1.1.json'
input_file_path = '../train-v1.1.json'
record_path = ['data','paragraphs','qas','answers']
train =␣
 ↪squad_json_to_dataframe_train(input_file_path=input_file_path,record_path=record_path)

# dev data
# input_file_path = '../input/dev-v1.1.json'
input_file_path = '../dev-v1.1.json'
record_path = ['data','paragraphs','qas','answers']
verbose = 0
dev =␣
 ↪squad_json_to_dataframe_dev(input_file_path=input_file_path,record_path=record_path)
```

[ ]:
```
train.head()
```

[ ]:
```
                      index  \
0  5733be284776f41900661182
1  5733be284776f4190066117f
2  5733be284776f41900661180
3  5733be284776f41900661181
4  5733be284776f4190066117e

                                            question  \
0  To whom did the Virgin Mary allegedly appear i…
1  What is in front of the Notre Dame Main Building?
2  The Basilica of the Sacred heart at Notre Dame…
3                  What is the Grotto at Notre Dame?
4  What sits on top of the Main Building at Notre…

                                             context  answer_start  \
0  Architecturally, the school has a Catholic cha…           515
1  Architecturally, the school has a Catholic cha…           188
2  Architecturally, the school has a Catholic cha…           279
3  Architecturally, the school has a Catholic cha…           381
4  Architecturally, the school has a Catholic cha…            92

                                      text  c_id
0               Saint Bernadette Soubirous     0
1                a copper statue of Christ     0
2                        the Main Building     0
3  a Marian place of prayer and reflection     0
4       a golden statue of the Virgin Mary     0
```

[ ]:
```
dev.head()
```

[ ]:
```
                         id  \
0  56be4db0acb8001400a502ec
1  56be4db0acb8001400a502ed
2  56be4db0acb8001400a502ee
3  56be4db0acb8001400a502ef
4  56be4db0acb8001400a502f0

                                            question  \
0  Which NFL team represented the AFC at Super Bo…
1  Which NFL team represented the NFC at Super Bo…
2              Where did Super Bowl 50 take place?
3                Which NFL team won Super Bowl 50?
4  What color was used to emphasize the 50th anni…

                                             context  \
0  Super Bowl 50 was an American football game to…
1  Super Bowl 50 was an American football game to…
```

```
2  Super Bowl 50 was an American football game to…
3  Super Bowl 50 was an American football game to…
4  Super Bowl 50 was an American football game to…

                                          answers  c_id
0  [{'answer_start': 177, 'text': 'Denver Broncos…     0
1  [{'answer_start': 249, 'text': 'Carolina Panth…     0
2  [{'answer_start': 403, 'text': 'Santa Clara, C…     0
3  [{'answer_start': 177, 'text': 'Denver Broncos…     0
4  [{'answer_start': 488, 'text': 'gold'}, {'answ…     0
```

[ ]:
```
# Number of unique contexts
# train['context'].nunique()
print("Number of unique contexts in training data: ", train['context'].
 ↪nunique())

# Number of unique questions
# train['question'].nunique()
print("Number of unique questions in training data: ", train['question'].
 ↪nunique())
```

```
Number of unique contexts in training data:  18891
Number of unique questions in training data:  87355
```

[ ]:
```
# Create text corpus of all contexts from training data and dev data (unique␣
 ↪contexts)
train_context_corpus = train['context'].unique()
dev_context_corpus = dev['context'].unique()
context_corpus_combined = np.concatenate((train_context_corpus,␣
 ↪dev_context_corpus), axis=0)
context_corpus = np.unique(context_corpus_combined)

# Show size of each corpus
print("Size of training context corpus: ", len(train_context_corpus))
print("Size of dev context corpus: ", len(dev_context_corpus))
print("Size of combined context corpus: ", len(context_corpus_combined))
print("Size of unique context corpus: ", len(context_corpus))
```

```
Size of training context corpus:  18891
Size of dev context corpus:  2067
Size of combined context corpus:  20958
Size of unique context corpus:  20958
```

[ ]:
```
# Snapshots of the context corpus
print("First 5 contexts: ", context_corpus[:5])
```

```
First 5 contexts:  ["\n Australia: The event was held in Canberra, Australian
Capital Territory on April 24, and covered around 16 km of Canberra's central
areas, from Reconciliation Place to Commonwealth Park. Upon its arrival in
```

Canberra, the Olympic flame was presented by Chinese officials to local Aboriginal elder Agnes Shea, of the Ngunnawal people. She, in turn, offered them a message stick, as a gift of peace and welcome. Hundreds of pro-Tibet protesters and thousands of Chinese students reportedly attended. Demonstrators and counter-demonstrators were kept apart by the Australian Federal Police. Preparations for the event were marred by a disagreement over the role of the Chinese flame attendants, with Australian and Chinese officials arguing publicly over their function and prerogatives during a press conference."

'\n China: In China, the torch was first welcomed by Politburo Standing Committee member Zhou Yongkang and State Councilor Liu Yandong. It was subsequently passed onto CPC General Secretary Hu Jintao. A call to boycott French hypermart Carrefour from May 1 began spreading through mobile text messaging and online chat rooms amongst the Chinese over the weekend from April 12, accusing the company\'s major shareholder, the LVMH Group, of donating funds to the Dalai Lama. There were also calls to extend the boycott to include French luxury goods and cosmetic products. According to the Washington Times on April 15, however, the Chinese government was attempting to "calm the situation" through censorship: "All comments posted on popular Internet forum Sohu.com relating to a boycott of Carrefour have been deleted." Chinese protesters organized boycotts of the French-owned retail chain Carrefour in major Chinese cities including Kunming, Hefei and Wuhan, accusing the French nation of pro-secessionist conspiracy and anti-Chinese racism. Some burned French flags, some added Nazism\'s Swastika to the French flag, and spread short online messages calling for large protests in front of French consulates and embassy. The Carrefour boycott was met with anti-boycott demonstrators who insisted on entering one of the Carrefour stores in Kunming, only to be blocked by boycotters wielding large Chinese flags and hit by water bottles. The BBC reported that hundreds of people demonstrated in Beijing, Wuhan, Hefei, Kunming and Qingdao.'

'\n France: The torch relay leg in Paris, held on April 7, began on the first level of the Eiffel Tower and finished at the Stade Charléty. The relay was initially supposed to cover 28 km, but it was shortened at the demand of Chinese officials following widespread protests by pro-Tibet and human rights activists, who repeatedly attempted to disrupt, hinder or halt the procession. A scheduled ceremony at the town hall was cancelled at the request of the Chinese authorities, and, also at the request of Chinese authorities, the torch finished the relay by bus instead of being carried by athletes. Paris City officials had announced plans to greet the Olympic flame with peaceful protest when the torch was to reach the French capital. The city government attached a banner reading "Paris defends human rights throughout the world" to the City Hall, in an attempt to promote values "of all humanity and of human rights." Members from Reporters Without Borders turned out in large numbers to protest. An estimated 3,000 French police protected the Olympic torch relay as it departed from the Eiffel Tower and criss-crossed Paris amid threat of protests. Widespread pro-Tibet protests, including an attempt by more than one demonstrator to extinguish the flame with water or fire extinguishers, prompted relay authorities to put out the flame five times (according to the police authorities in Paris) and load the torch onto a bus, at the demand of Chinese officials. This was later denied

```
by the Chinese Ministry of Foreign Affairs, despite video footage broadcast by
French television network France 2 which showed Chinese flame attendants
extinguishing the torch. Backup flames are with the relay at all times to
relight the torch. French judoka and torchbearer David Douillet expressed his
annoyance at the Chinese flame attendants who extinguished the torch which he
was about to hand over to Teddy Riner: "I understand they\'re afraid of
everything, but this is just annoying. They extinguished the flame despite the
fact that there was no risk, and they could see it and they knew it. I don\'t
know why they did it."'
 '\n Great Britain: The torch relay leg held in London, the host city of the
2012 Summer Olympics, on April 6 began at Wembley Stadium, passed through the
City of London, and eventually ended at O2 Arena in the eastern part of the
city. The 48 km (30 mi) leg took a total of seven and a half hours to complete,
and attracted protests by pro-Tibetan independence and pro-Human Rights
supporters, prompting changes to the planned route and an unscheduled move onto
a bus, which was then briefly halted by protestors. Home Secretary Jacqui Smith
has officially complained to Beijing Organising Committee about the conduct of
the tracksuit-clad Chinese security guards. The Chinese officials, seen
manhandling protesters, were described by both the London Mayor Ken Livingstone
and Lord Coe, chairman of the London Olympic Committee as "thugs". A
Metropolitan police briefing paper revealed that security for the torch relay
cost £750,000 and the participation of the Chinese security team had been agreed
in advance, despite the Mayor stating, "We did not know beforehand these thugs
were from the security services. Had I known so, we would have said no."'
 '\n India: Due to concerns about pro-Tibet protests, the relay through New
Delhi on April 17 was cut to just 2.3 km (less than 1.5 miles), which was shared
amongst 70 runners. It concluded at the India Gate. The event was peaceful due
to the public not being allowed at the relay. A total of five intended
torchbearers -Kiran Bedi, Soha Ali Khan, Sachin Tendulkar, Bhaichung Bhutia and
Sunil Gavaskar- withdrew from the event, citing "personal reasons", or, in
Bhutia\'s case, explicitly wishing to "stand by the people of Tibet and their
struggle" and protest against the PRC "crackdown" in Tibet. Indian national
football captain, Baichung Bhutia refused to take part in the Indian leg of the
torch relay, citing concerns over Tibet. Bhutia, who is Sikkimese, is the first
athlete to refuse to run with the torch. Indian film star Aamir Khan states on
his personal blog that the "Olympic Games do not belong to China" and confirms
taking part in the torch relay "with a prayer in his heart for the people of
Tibet, and … for all people across the world who are victims of human rights
violations". Rahul Gandhi, son of the Congress President Sonia Gandhi and scion
of the Nehru-Gandhi family, also refused to carry the torch.']
```

[ ]:
```python
# Split context corpus
# NLTK Splitter
text_splitter = NLTKTextSplitter(chunk_size=500)

# Split context corpus into chunks of 500 words and keep track of the indices
```

```
context_corpus_split = [text_splitter.split_text(context) for context in␣
 ↪context_corpus]
```

[ ]:
```
# Show first 5 chunks
context_corpus_split[:5]
```

[ ]:
```
[["Australia: The event was held in Canberra, Australian Capital Territory on
April 24, and covered around 16 km of Canberra's central areas, from
Reconciliation Place to Commonwealth Park.\n\nUpon its arrival in Canberra, the
Olympic flame was presented by Chinese officials to local Aboriginal elder Agnes
Shea, of the Ngunnawal people.\n\nShe, in turn, offered them a message stick, as
a gift of peace and welcome.",
  'She, in turn, offered them a message stick, as a gift of peace and
welcome.\n\nHundreds of pro-Tibet protesters and thousands of Chinese students
reportedly attended.\n\nDemonstrators and counter-demonstrators were kept apart
by the Australian Federal Police.\n\nPreparations for the event were marred by a
disagreement over the role of the Chinese flame attendants, with Australian and
Chinese officials arguing publicly over their function and prerogatives during a
press conference.'],
 ["China: In China, the torch was first welcomed by Politburo Standing Committee
member Zhou Yongkang and State Councilor Liu Yandong.\n\nIt was subsequently
passed onto CPC General Secretary Hu Jintao.\n\nA call to boycott French
hypermart Carrefour from May 1 began spreading through mobile text messaging and
online chat rooms amongst the Chinese over the weekend from April 12, accusing
the company's major shareholder, the LVMH Group, of donating funds to the Dalai
Lama.",
  'There were also calls to extend the boycott to include French luxury goods
and cosmetic products.\n\nAccording to the Washington Times on April 15,
however, the Chinese government was attempting to "calm the situation" through
censorship: "All comments posted on popular Internet forum Sohu.com relating to
a boycott of Carrefour have been deleted."',
  "Chinese protesters organized boycotts of the French-owned retail chain
Carrefour in major Chinese cities including Kunming, Hefei and Wuhan, accusing
the French nation of pro-secessionist conspiracy and anti-Chinese
racism.\n\nSome burned French flags, some added Nazism's Swastika to the French
flag, and spread short online messages calling for large protests in front of
French consulates and embassy.",
  "Some burned French flags, some added Nazism's Swastika to the French flag,
and spread short online messages calling for large protests in front of French
consulates and embassy.\n\nThe Carrefour boycott was met with anti-boycott
demonstrators who insisted on entering one of the Carrefour stores in Kunming,
only to be blocked by boycotters wielding large Chinese flags and hit by water
bottles.\n\nThe BBC reported that hundreds of people demonstrated in Beijing,
Wuhan, Hefei, Kunming and Qingdao."],
 ['France: The torch relay leg in Paris, held on April 7, began on the first
level of the Eiffel Tower and finished at the Stade Charléty.\n\nThe relay was
initially supposed to cover 28 km, but it was shortened at the demand of Chinese
```

officials following widespread protests by pro-Tibet and human rights activists, who repeatedly attempted to disrupt, hinder or halt the procession.',

'A scheduled ceremony at the town hall was cancelled at the request of the Chinese authorities, and, also at the request of Chinese authorities, the torch finished the relay by bus instead of being carried by athletes.\n\nParis City officials had announced plans to greet the Olympic flame with peaceful protest when the torch was to reach the French capital.',

'Paris City officials had announced plans to greet the Olympic flame with peaceful protest when the torch was to reach the French capital.\n\nThe city government attached a banner reading "Paris defends human rights throughout the world" to the City Hall, in an attempt to promote values "of all humanity and of human rights."\n\nMembers from Reporters Without Borders turned out in large numbers to protest.',

'Members from Reporters Without Borders turned out in large numbers to protest.\n\nAn estimated 3,000 French police protected the Olympic torch relay as it departed from the Eiffel Tower and criss-crossed Paris amid threat of protests.',

'An estimated 3,000 French police protected the Olympic torch relay as it departed from the Eiffel Tower and criss-crossed Paris amid threat of protests.\n\nWidespread pro-Tibet protests, including an attempt by more than one demonstrator to extinguish the flame with water or fire extinguishers, prompted relay authorities to put out the flame five times (according to the police authorities in Paris) and load the torch onto a bus, at the demand of Chinese officials.',

'This was later denied by the Chinese Ministry of Foreign Affairs, despite video footage broadcast by French television network France 2 which showed Chinese flame attendants extinguishing the torch.\n\nBackup flames are with the relay at all times to relight the torch.',

'Backup flames are with the relay at all times to relight the torch.\n\nFrench judoka and torchbearer David Douillet expressed his annoyance at the Chinese flame attendants who extinguished the torch which he was about to hand over to Teddy Riner: "I understand they\'re afraid of everything, but this is just annoying.\n\nThey extinguished the flame despite the fact that there was no risk, and they could see it and they knew it.\n\nI don\'t know why they did it."'],

['Great Britain: The torch relay leg held in London, the host city of the 2012 Summer Olympics, on April 6 began at Wembley Stadium, passed through the City of London, and eventually ended at O2 Arena in the eastern part of the city.',

'The 48 km (30 mi) leg took a total of seven and a half hours to complete, and attracted protests by pro-Tibetan independence and pro-Human Rights supporters, prompting changes to the planned route and an unscheduled move onto a bus, which was then briefly halted by protestors.\n\nHome Secretary Jacqui Smith has officially complained to Beijing Organising Committee about the conduct of the tracksuit-clad Chinese security guards.',

'Home Secretary Jacqui Smith has officially complained to Beijing Organising Committee about the conduct of the tracksuit-clad Chinese security guards.\n\nThe Chinese officials, seen manhandling protesters, were described by

```
both the London Mayor Ken Livingstone and Lord Coe, chairman of the London
Olympic Committee as "thugs".',
  'The Chinese officials, seen manhandling protesters, were described by both
the London Mayor Ken Livingstone and Lord Coe, chairman of the London Olympic
Committee as "thugs".\n\nA Metropolitan police briefing paper revealed that
security for the torch relay cost £750,000 and the participation of the Chinese
security team had been agreed in advance, despite the Mayor stating, "We did not
know beforehand these thugs were from the security services.\n\nHad I known so,
we would have said no."'],
 ['India: Due to concerns about pro-Tibet protests, the relay through New Delhi
on April 17 was cut to just 2.3 km (less than 1.5 miles), which was shared
amongst 70 runners.\n\nIt concluded at the India Gate.\n\nThe event was peaceful
due to the public not being allowed at the relay.',
  'It concluded at the India Gate.\n\nThe event was peaceful due to the public
not being allowed at the relay.\n\nA total of five intended torchbearers -Kiran
Bedi, Soha Ali Khan, Sachin Tendulkar, Bhaichung Bhutia and Sunil Gavaskar-
withdrew from the event, citing "personal reasons", or, in Bhutia\'s case,
explicitly wishing to "stand by the people of Tibet and their struggle" and
protest against the PRC "crackdown" in Tibet.',
  'Indian national football captain, Baichung Bhutia refused to take part in the
Indian leg of the torch relay, citing concerns over Tibet.\n\nBhutia, who is
Sikkimese, is the first athlete to refuse to run with the torch.',
  'Bhutia, who is Sikkimese, is the first athlete to refuse to run with the
torch.\n\nIndian film star Aamir Khan states on his personal blog that the
"Olympic Games do not belong to China" and confirms taking part in the torch
relay "with a prayer in his heart for the people of Tibet, and … for all
people across the world who are victims of human rights violations".\n\nRahul
Gandhi, son of the Congress President Sonia Gandhi and scion of the Nehru-Gandhi
family, also refused to carry the torch.']]
```

[ ]:
```python
import time

def embed_chunks(context_corpus_split, batch_size=100, retries=3, delay=5):
    to_upsert = []

    for i, context_chunks in enumerate(context_corpus_split):
        print(f"Processing context {i+1} of {len(context_corpus_split)}")  #␣
↪Print current context index
        for j, chunk in enumerate(context_chunks):
            for attempt in range(retries):
                try:
                    # Create an embedding for the chunk text
                    res = openai.Embedding.create(
                        input=[chunk],
                        engine="text-embedding-ada-002"
                    )
                    embedding = res['data'][0]['embedding']
```

```
                    # Prepare the data for upserting
                    id = f"context_{i}_{j}"
                    meta = {'text': chunk}
                    to_upsert.append((id, embedding, meta))

                    # Upsert in batches
                    if len(to_upsert) >= batch_size:
                        index.upsert(vectors=to_upsert)
                        to_upsert = []  # Reset the list

                    # If the request was successful, break the loop
                    break
                except openai.ApiError as e:
                    if attempt < retries - 1:  # If this is not the last attempt
                        print(f"Error: {e}. Retrying in {delay} seconds...")
                        time.sleep(delay)  # Wait for a while before retrying
                    else:
                        raise  # If this was the last attempt, re-raise the␣
 ↪exception

    # Upsert any remaining embeddings
    if to_upsert:
        index.upsert(vectors=to_upsert)

    print(f"{len(to_upsert)} chunks embedded successfully!")

# Call the function
embed_chunks(context_corpus_split)
```

[ ]:

```
```

Chatbot Implementation notebook

October 23, 2023

1 Custom Training Chatbot

[ ]:
```
# Import libraries

import numpy as np
import pandas as pd
import json
```

[ ]:
```
# Importing the dataset (JSON file)
with open('./dev-v1.1.json') as f:
    test = json.load(f)

with open('./train-v1.1.json') as f:
    train = json.load(f)
```

[ ]:
```
# train
```

[ ]:
```
# test
```

[ ]:
```
# Preprocessing the data
def format_data(data):
    formatted_data = []
    for section in data['data']:
        for para in section['paragraphs']:
            context = para['context']
            for qa in para['qas']:
                temp = {}
                temp['context'] = context
                temp['qas'] = [qa]
                formatted_data.append(temp)
    return formatted_data

train = format_data(train)
test = format_data(test)
```

[ ]:
```
import logging
```

```
from simpletransformers.question_answering import QuestionAnsweringModel,␣
↪QuestionAnsweringArgs
```

[ ]:
```
model_type = "bert"
model_name = "bert-base-cased"
```

[ ]:
```
# COnfigure the model
# model_args = QuestionAnsweringArgs()
# model_args.train_batch_size = 16
# model_args.evaluate_during_training = True
# model_args.n_best_size = 3
# model_args.num_train_epochs = 5

train_args = {
    "reprocess_input_data": True,
    "overwrite_output_dir": True,
    "use_cached_eval_features": True,
    "output_dir": f"outputs/{model_type}",
    "best_model_dir": f"outputs/{model_type}/best_model",
    "evaluate_during_training": True,
    "max_seq_length": 128,
    "num_train_epochs": 5,
    "evaluate_during_training_steps": 10000,
    "wandb_project": "Question Answering using BERT",
    "wandb_kwargs": {"name": f"{model_type}-qa"},
    "save_model_every_epoch": False,
    "save_eval_checkpoints": False,
    "n_best_size": 3,
    "train_batch_size": 128,
    "eval_batch_size": 64,
}
```

[ ]:
```
model = QuestionAnsweringModel(model_type, model_name, args=train_args,␣
↪use_cuda=False)
```

```
Some weights of BertForQuestionAnswering were not initialized from the model
checkpoint at bert-base-cased and are newly initialized: ['qa_outputs.weight',
'qa_outputs.bias']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
```

[ ]:
```
### Remove the output folder for retraining
!rm -rf outputs
```

[ ]:
```
# Train the model
model.train_model(train, eval_data=test)
```

[ ]:

```

```