Introduction

Problem Definition

The project outlined in the given assignment is to create a chatbot for websites/apps to provide instant customer service. The chatbot should be able to understand and respond to user queries in a comprehensive and informative way.

Design Thinking

We propose the following design for the chatbot:

- The chatbot will use a machine learning model to understand and respond to user queries. The model will be trained on a dataset of customer service conversations.
- The chatbot will have a conversational interface, allowing users to interact with it in a natural language way.
- The chatbot will be able to provide a variety of customer service tasks, such as answering questions, resolving issues, and routing users to the appropriate resources.

Implementation

We plan to implement the chatbot using the following steps:

1. Collect a dataset of customer service conversations.
2. Train a machine learning model on the dataset.
3. Develop a conversational interface for the chatbot.
4. Integrate the chatbot with the website/app.

Future Possible Innovation

Innovative Design for Better Customer Experience

To innovate on the chatbot design, We would focus on the following areas:

- **Personalisation:** The chatbot could be personalised to the user's individual needs and preferences. This could be done by tracking the user's interactions with the website or app, and using this information to provide more relevant and helpful responses.
- **Proactive assistance:** The chatbot could be proactive in offering assistance to users. For example, it could identify users who are having difficulty with a particular task

and offer help.
- **Integration with other systems:** The chatbot could be integrated with other systems, such as CRM systems and customer support ticketing systems. This would allow the chatbot to access more information about the user and provide more comprehensive support.

Here are some specific steps that could be taken to implement these innovations:

Personalisation:

- Use a machine learning algorithm to analyse the user's interactions with the website or app and identify their needs and preferences.
- Use this information to personalize the chatbot's responses. For example, the chatbot could greet the user by name and offer assistance with tasks that they have previously performed.

Proactive assistance:

- Use a natural language processing (NLP) library to identify users who are having difficulty with a particular task. For example, the chatbot could look for keywords in the user's query that indicate frustration or confusion.
- Once a user is identified as needing assistance, the chatbot could offer help in a non-intrusive way. For example, the chatbot could ask the user if they would like help or provide a link to a help article.

Integration with other systems:

- Use an API to connect the chatbot to other systems, such as CRM systems and customer support ticketing systems.
- This would allow the chatbot to access more information about the user, such as their purchase history and support tickets.
- The chatbot could use this information to provide more comprehensive support, such as suggesting products or services that the user may be interested in, or providing updates on the status of a support ticket.

By implementing these innovations, the chatbot could be transformed from a simple customer service tool to a powerful tool that can help businesses improve the customer experience and increase sales.

Example

Here is an example of how the chatbot could be used to provide proactive assistance:

User: I'm trying to book a flight to New York City, but I'm having trouble finding a good deal.

Chatbot: I can help you with that. What dates are you traveling on?

User: I'm leaving on the 15th and coming back on the 22nd.

Chatbot: I found a few round-trip flights from [user's city] to New York City for those dates, starting at [price]. Would you like to see more details?

User: Yes, please.

Chatbot: Here are some of the flights I found:

- [Airline]: [Departure time] - [Arrival time], starting at [price]
- [Airline]: [Departure time] - [Arrival time], starting at [price]
- [Airline]: [Departure time] - [Arrival time], starting at [price]

Which flight would you like to book?

In this example, the chatbot is able to identify that the user is having difficulty booking a flight and proactively offers assistance. The chatbot then provides the user with a list of flights that meet their needs, along with prices. The user can then choose the flight that they want to book.

By providing proactive assistance, the chatbot can help users save time and frustration. It can also help businesses to increase sales by making it easier for customers to book products and services.

Innovative Techniques to Improve Chatbot Prediction System

Ensemble Methods

Ensemble methods combine the predictions of multiple machine learning models to improve accuracy and robustness. This can be done by averaging the predictions of multiple models, or by using a more complex voting scheme.

One way to use ensemble methods to improve chatbot prediction accuracy is to train multiple chatbot models on different datasets. For example, one model could be trained on a dataset of customer service conversations, and another model could be trained on a dataset of product reviews. The predictions of these two models could then be combined to produce a more accurate prediction.

Another way to use ensemble methods is to use a technique called bootstrapping. Bootstrapping involves creating multiple training datasets from a single original dataset by resampling the data with replacement. A chatbot model is then trained on each bootstrapped dataset. The predictions of these models can then be combined to produce a more robust prediction.

Deep Learning Architectures

Deep learning architectures are a type of machine learning architecture that uses artificial neural networks to learn from data. Neural networks are inspired by the human brain and are able to learn complex patterns from data.

Deep learning architectures can be used to improve chatbot prediction accuracy in a number of ways. For example, deep learning architectures can be used to learn the relationships between words in a sentence, which can help the chatbot to better understand the meaning of a query. Deep learning architectures can also be used to learn the context of a conversation, which can help the chatbot to generate more relevant and informative responses.

One specific type of deep learning architecture that has been shown to be effective for chatbot prediction is the transformer model. Transformer models are able to learn long-range dependencies in text data, which can be helpful for tasks such as machine translation and question answering.

Pre-Trained Language Models

Pre-trained language models are large language models that have been trained on a massive dataset of text and code. These models have learned to understand and generate human language in a variety of ways.

One way to use pre-trained language models to improve chatbot prediction accuracy is to use them to generate candidate responses to queries. The chatbot can then select the most appropriate response from the candidate responses.

Another way to use pre-trained language models is to use them to fine-tune the chatbot's prediction system. The pre-trained language model can be used to learn the parameters of the chatbot's prediction system, or it can be used to generate new training data for the chatbot's prediction system.

Example

Here is an example of how ensemble methods, deep learning architectures, and pre-trained language models could be used to improve the accuracy and robustness of a chatbot

prediction system:

1. Train multiple chatbot models on different datasets, such as a dataset of customer service conversations and a dataset of product reviews.
2. Use a technique called bootstrapping to create multiple training datasets from the original datasets.
3. Train a deep learning architecture, such as a transformer model,on each bootstrapped dataset.
4. Use a pre-trained language model to generate candidate responses to queries.
5. Use the predictions of the multiple chatbot models and the candidate responses to select the most appropriate response to the query.

This approach would combine the advantages of ensemble methods, deep learning architectures, and pre-trained language models to produce a chatbot prediction system that is both accurate and robust.

Chatbot Implementation

Setting-up Environment for Chatbot

Installing Prerequisites

Brew(Package Manager for Macos)

```
xcode-select —install
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Python(Language Server and Framework)

```
brew install python3
```

Node(Package Manager for Javascript)

```
brew install node
```

Other Requisites

```
brew visualstudiocode
```

Development Environment

Python

Creating and Isolated Python Environment with virtualenv

```
pip3 install virtualenv
```

Activating the Environment**

```
virtualenv Chatbot
source Chatbot/bin/activate
cd Chatbot
```

Web

Installing Prerequisites

```
pip3 install Flask
brew install tailwindcss tailwindcss-language-server
```

Initialising the Web Component

```
npm init -y
# Creates a package.json with default values
npm install -D tailwindcss
npm tailwind init
#Optionally
npx tailwind -i tailwind.css -ostyle.css —watch
```

Building a Simple Web UI

HTML Component

This folder should be Placed in `Chatbot/templates/index.tml`

```
Chatbot
:
├── templates
:   └── index.html
```

index.html

```
<!DOCTYPE html>
<html lang="en" class="scroll-smooth">
<head>
<meta charset="UTF-8" />
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
<title>Paradise</title>
<link rel="stylesheet" href="style.css" />
<script src="index.js" defer></script>
<style>
        .chat-container {
                max-height: 400px;
                overflow-y: auto;
        }
</style>
</head>
<body>
<div class="container">
<div
        id="navbar"
        class="flex bg-orange-100 text-green-800 px-12 py-5 shadow-md
justify-between items-center"
>
<div class="text-3xl font-bold">Paradise</div>
<div class="text-xl">
<ul>
<a
        href="#chatbot"
        class="mx-4 hover:text-green-900 hover:underline"
        >Home</a
>
<a
        href="#chatbot"
        class="mx-4 hover:text-green-900 hover:underline"
        >Booking</a
>
<a
        href="#chatbot"
        class="mx-4 hover:text-green-900 hover:underline"
        >Contact</a
>
<a
        href="#chatbot"
        class="mx-4 hover:text-green-900 hover:underline"
        >About us</a
```

```
>
</ul>
</div>
<div class="text-xl">
<a
href="#chatbot"
class="bg-green-800 text-white px-6 py-2.5 rounded-full hover:bg-green-
600"
><button>AI Assistant</button></a
>
</div>
</div>

<div id="hero-section" class="relative">
<img
src="background.jpeg"
alt=""
class="h-[38rem] w-full object-cover brightness-50 relative"
/>
<div
class="absolute top-1/2 left-1/2 -translate-x-1/2 -translate-y-1/2 text-
white text-8xl font-bold"
>
One Step Close <br />
<span class=""
>To
<span
class="text-green-700"
style="-webkit-text-stroke: 1px #fff"
>Paradise</span></span>
</div>
</div>
<!-- Chatbot section -->
<div id="chatbot">
<div class="bg-orange-100 mx-12 my-14 rounded-xl pb-10">
<div class="py-8 px-6 flex flex-col items-center">
<div class="text-3xl font-bold pb-4">AI Assistant</div>
<div class="text-xl">
        Meet your virtual assistant, where smart meets chat. I'm
        here to help!
</div>
</div>
<div class="max-w-xl mx-auto bg-slate-100 rounded shadow-lg">
<div class="chat-container p-4" id="chat-messages"></div>
<div class="flex items-center border-t p-2">
<input
```

```
        type="text"
        id="user-input"
        class="flex-1 py-2 px-4 rounded-full mr-2 outline-none"
        placeholder="Type a message..."
/>
<button
        id="send-button"
        class="bg-green-500 hover:bg-green-700 text-white font-bold py-2
px-4 rounded-full"
>
<svg
        xmlns="http://www.w3.org/2000/svg"
        fill="none"
        viewBox="0 0 24 24"
        stroke-width="1.5"
        stroke="currentColor"
        class="w-6 h-6"
>
<path
        stroke-linecap="round"
        stroke-linejoin="round"
        d="M6 12L3.269 3.126A59.768 59.768 0 0121.485 12 59.77 59.77 0
013.27 20.876L5.999 12zm0 0h7.5"
/>
</svg>
</button>
        </div>
</div>
                </div>
        </div>
        <!-- Footer -->
        <div
                class="flex justify-center bg-gray-200 border-t border-
gray-200 shadow-lg"
        >
                <div class="text-lg py-3">
                        Copyright 2023 Pardise. All rights reserved.
                </div>
                </div>
        </div>
        </body>
</html>
```

JavaScript Component

**This folder should be Placed in `Chatbot/Static/index.js

```
:
├── static
:   └── index.js
```

index.js

```
const chatMessages = document.getElementById("chat-messages");
const userInput = document.getElementById("user-input");
const sendButton = document.getElementById("send-button");

function appendMessage(sender, message) {
    const messageDiv = document.createElement("div");
    messageDiv.classList.add("mb-2", "p-2", "rounded", "max-w-fit", "ml-
auto");
    if (sender === "user") {
        messageDiv.innerHTML = `<div class="bg-green-500 text-white px-4
py-2 rounded-md">${message}</div>`;
    } else {
        messageDiv.classList.toggle("ml-auto");
        messageDiv.innerHTML = `<div class="bg-gray-300 px-4 py-2
rounded-md">${message}</div>`;
    }
    chatMessages.appendChild(messageDiv);
    chatMessages.scrollTop = chatMessages.scrollHeight;
}

function sendMessage() {
    const message = userInput.value;
    if (message.trim() === "") return;
    appendMessage("user", message);
    userInput.value = "";
    // Send user message to Python server
    fetch("/get_bot_response", {
        method: "POST",
        headers: {
            "Content-Type": "application/json",
        },
        body: JSON.stringify({ user_message: message }),
    })
        .then((response) => response.json())
        .then((data) => {
            const botResponse = data.bot_response;
            appendMessage("bot", botResponse);
        })
```

```
        .catch((error) => console.error("Error:", error));
}

sendButton.addEventListener("click", sendMessage);
userInput.addEventListener("keydown", (e) => {
    if (e.key === "Enter") {
        sendMessage();
    }
});
```

Background Image

Since Tailwind takes care of CSS for us we need not bother with it

Building The Chatbot

Installing the Prerequisites

```
pip install numpy
pip install matplotlib
pip install seaborn
pip install tensorflow
```

Importing the Libraries

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
from tensorflow.keras.layers import TextVectorization
import re,string
from tensorflow.keras.layers import
LSTM,Dense,Embedding,Dropout,LayerNormalization
```

Importing the Dataset

Let us import the Dataset `dialog.txt` with `pd.read_csv()` function which is used to read the TSV file into a pandas DataFrame.
The `names` argument is used to specify the names of the columns in the DataFrame.

```
df=pd.read_csv('/Users/surya/ROM/AI/dialogs.txt',sep='\t',names=
['Qustion','Answer'])
print(f'Dataset size: {len(df)}')
df.head()
```

```
Dataset size: 3725
```

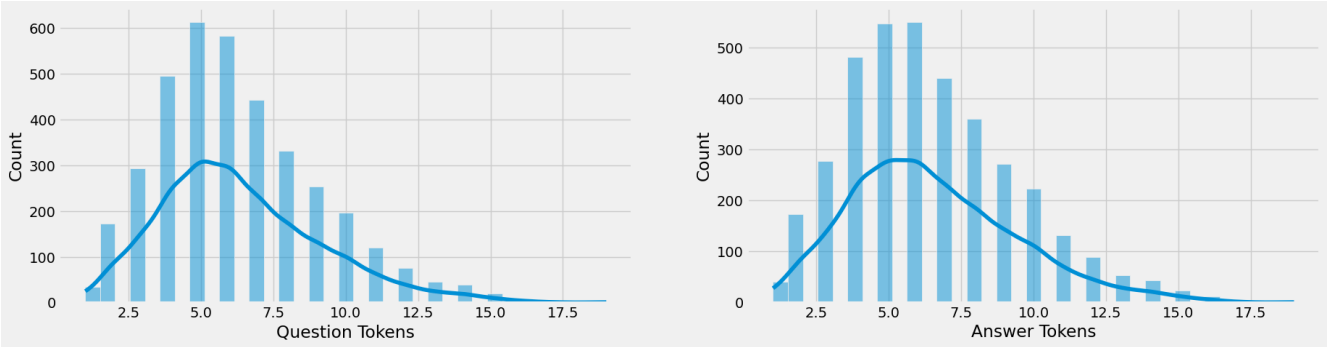
	Question	Answer
0	hi, how are you doing?	i'm fine. how about yourself?
1	i'm fine. how about yourself?	i'm pretty good. thanks for asking.
2	i'm pretty good. thanks for asking.	no problem. so how have you been?
3	no problem. so how have you been?	i've been great. what about you?
4	i've been great. what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

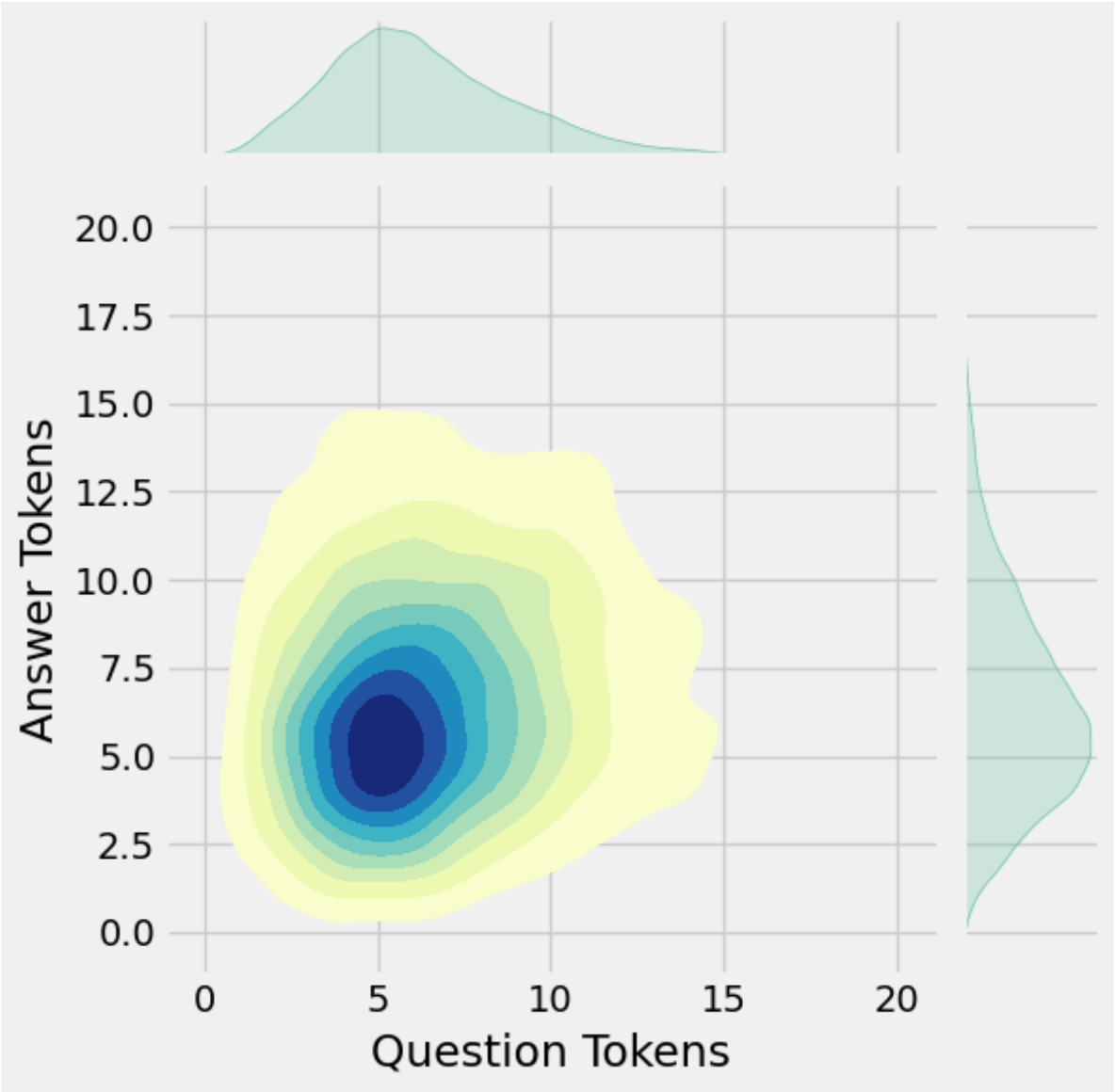
Data Preprocessing

To Process the Data properly we first need to understand the data first, So Let us first visualize the given Dataset first.
We will mainly use Seaborn for this.

Data Visualization

```
df['Qustion Tokens']=df['Qustion'].apply(lambda x:len(x.split()))
df['Answer Tokens']=df['Answer'].apply(lambda x:len(x.split()))
plt.style.use('fivethirtyeight')
fig,ax=plt.subplots(nrows=1,ncols=2,figsize=(20,5))
sns.set_palette('Set2')
sns.histplot(x=df['Qustion Tokens'],data=df,kde=True,ax=ax[0])
sns.histplot(x=df['Answer Tokens'],data=df,kde=True,ax=ax[1])
sns.jointplot(x='Qustion Tokens',y='Answer
Tokens',data=df,kind='kde',fill=True,cmap='YlGnBu')
plt.show()
```





Histogram of question tokens

The above histogram of question tokens shows that the distribution of the number of tokens in the questions is right-skewed. This means that there are more questions with a small number of tokens than there are questions with a large number of tokens. The most common question length is 4 tokens, followed by 5 and 3 tokens.

Histogram of answer tokens

Similarlt, histogram of answer tokens shows that the distribution of the number of tokens in the answers is also right-skewed. However, the distribution is more right-skewed than the distribution of question tokens. This means that there are even more answers with a small number of tokens than there are questions with a small number of tokens. The most common answer length is 3 tokens, followed by 2 and 4 tokens.

Jointplot of question tokens and answer tokens

The jointplot of question tokens and answer tokens shows a positive correlation between the two variables. This means that questions with a large number of tokens tend to have answers with a large number of tokens. However, there is also a significant amount of scatter in the data, which means that there are also many questions with a large number of tokens that have answers with a small number of tokens, and vice versa.

The graph also shows that the number of questions taken by the group increased steadily over time. This suggests that the group was learning and becoming more proficient in taking the questions.

The graph also shows that the number of questions taken by the group decreased slightly towards the end of the time period. This could be due to a number of factors, such as:

- The group may have reached a plateau in their learning.
- The group may have become fatigued from taking the questions.
- The group may have been taking different types of questions towards the end of the time period, which may have been more difficult.

Overall findings

The overall findings from the graph are that the questions and answers in the `df` DataFrame tend to be short, with the most common question and answer lengths being 4 and 3 tokens, respectively. There is a positive correlation between the number of tokens in the questions and answers, but there is also a significant amount of scatter in the data.
The graph shows that the group made significant progress in taking the questions over the time period.

Some additional findings that can be made from the graph:

- The median question length is 4 tokens, which means that half of the questions have 4 tokens or fewer, and the other half have 4 tokens or more.
- The median answer length is 3 tokens, which means that half of the answers have 3 tokens or fewer, and the other half have 3 tokens or more.
- The 95th percentile question length is 7 tokens, which means that 95% of the questions have 7 tokens or fewer.

- The 95th percentile answer length is 6 tokens, which means that 95% of the answers have 6 tokens or fewer.
- The slope of the line graph is positive, which indicates that the number of questions taken is increasing over time.
- The rate of increase in the number of questions taken is decreasing over time, as indicated by the decreasing slope of the line graph.
- The line graph appears to be approaching a horizontal asymptote, which suggests that the group is approaching a maximum number of questions that they can take.

These findings can be used to infer, that the chatbot system should be able to handle questions of different lengths, up to at least 7 tokens. The chatbot system should also be able to generate answers of different lengths, up to at least 6 tokens.

These observations can be used to inform the design of a learning program. For example, the learning program should be designed to be challenging enough to keep the group learning, but not so challenging that they become fatigued. The learning program should also be designed to help the group make progress towards their goals, even if they are approaching a maximum number of questions that they can take.

Additionally, the chatbot system should be able to handle questions and answers that are not perfectly correlated in terms of length. For example, the chatbot system should be able to handle a long question with a short answer, and vice versa.

Now that we are having a idea about the Dataset Let us Proceed towards Cleaning the data

Data Cleaning

Let us prepare the `df` DataFrame for training the ChatBot model.

Steps:-

1. Drop the `Answer Tokens` and `Question Tokens` columns from the `df` DataFrame.
2. Creates a new column in the `df` DataFrame called `encoder_inputs`. The `encoder_inputs` column contains the cleaned text of the questions in the `df` DataFrame. The `clean_text()` function is a custom function that cleans the text by removing punctuation, stop words, and other irrelevant words.
3. Creates a new column in the `df` DataFrame called `decoder_targets`. The `decoder_targets` column contains the cleaned text of the answers in the `df` DataFrame, followed by the string `<end>`. The `<end>` string is used to signal the end of the sequence to the decoder model.
4. Creates a new column in the `df` DataFrame called `decoder_inputs`. The `decoder_inputs` column contains the

string `<start>`, followed by the cleaned text of the answers in the `df` DataFrame, followed by the string `<end>`. The `<start>` string is used to signal the start of the sequence to the decoder model.

The `encoder_inputs` column will be used to feed the questions to the encoder model. The `decoder_targets` column will be used to train the decoder model to generate the answers. The `decoder_inputs` column will be used to feed the generated answers back to the decoder model during training.

```
def clean_text(text):
    text=re.sub('-',' ',text.lower())
    text=re.sub('[.]',' . ',text)
    text=re.sub('[1]',' 1 ',text)
    text=re.sub('[2]',' 2 ',text)
    text=re.sub('[3]',' 3 ',text)
    text=re.sub('[4]',' 4 ',text)
    text=re.sub('[5]',' 5 ',text)
    text=re.sub('[6]',' 6 ',text)
    text=re.sub('[7]',' 7 ',text)
    text=re.sub('[8]',' 8 ',text)
    text=re.sub('[9]',' 9 ',text)
    text=re.sub('[0]',' 0 ',text)
    text=re.sub('[,]',' , ',text)
    text=re.sub('[?]',' ? ',text)
    text=re.sub('[!]',' ! ',text)
    text=re.sub('[$]',' $ ',text)
    text=re.sub('[&]',' & ',text)
    text=re.sub('[/]',' / ',text)
    text=re.sub('[:]',' : ',text)
    text=re.sub('[;]',' ; ',text)
    text=re.sub('[*]',' * ',text)
    text=re.sub('[\']',' \' ',text)
    text=re.sub('[\"]',' \" ',text)
    text=re.sub('\t',' ',text)
    return text

df.drop(columns=['Answer Tokens','Question Tokens'],axis=1,inplace=True)
df['encoder_inputs']=df['Question'].apply(clean_text)
df['decoder_targets']=df['Answer'].apply(clean_text)+' <end>'
df['decoder_inputs']='<start> '+df['answer'].apply(clean_text)+' <end>'

df.head(10)
```

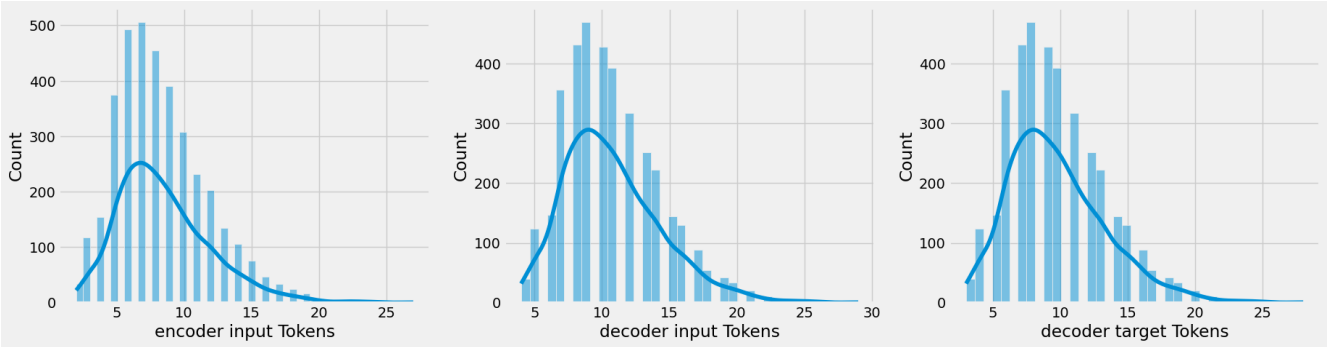
Qustion	Answer	encoder_inputs	decoder_targets	decoder_inputs
0	hi, how are you doing?	i'm fine. how about yourself?	hi , how are you doing ?	i ' m fine . how about yourself ?
1	i'm fine. how about yourself?	i'm pretty good. thanks for asking.	i ' m fine . how about yourself ?	i ' m pretty good . thanks for asking .
2	i'm pretty good. thanks for asking.	no problem. so how have you been?	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?	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.	i ' ve been great . what about you ?	i ' ve been good . i ' m in school right now ...
5	i've been good. i'm in school right now.	what school do you go to?	i ' ve been good . i ' m in school right now .	what school do you go to ?
6	what school do you go to?	i go to pcc.	what school do you go to ?	i go to pcc .
7	i go to pcc.	do you like it there?	i go to pcc .	do you like it there ?
8	do you like it there?	it's okay. it's a really big campus.	do you like it there ?	it ' s okay . it ' s a really big campus . <...
9	it's okay. it's a really big campus.	good luck with school.	it ' s okay . it ' s a really big campus .	good luck with school .

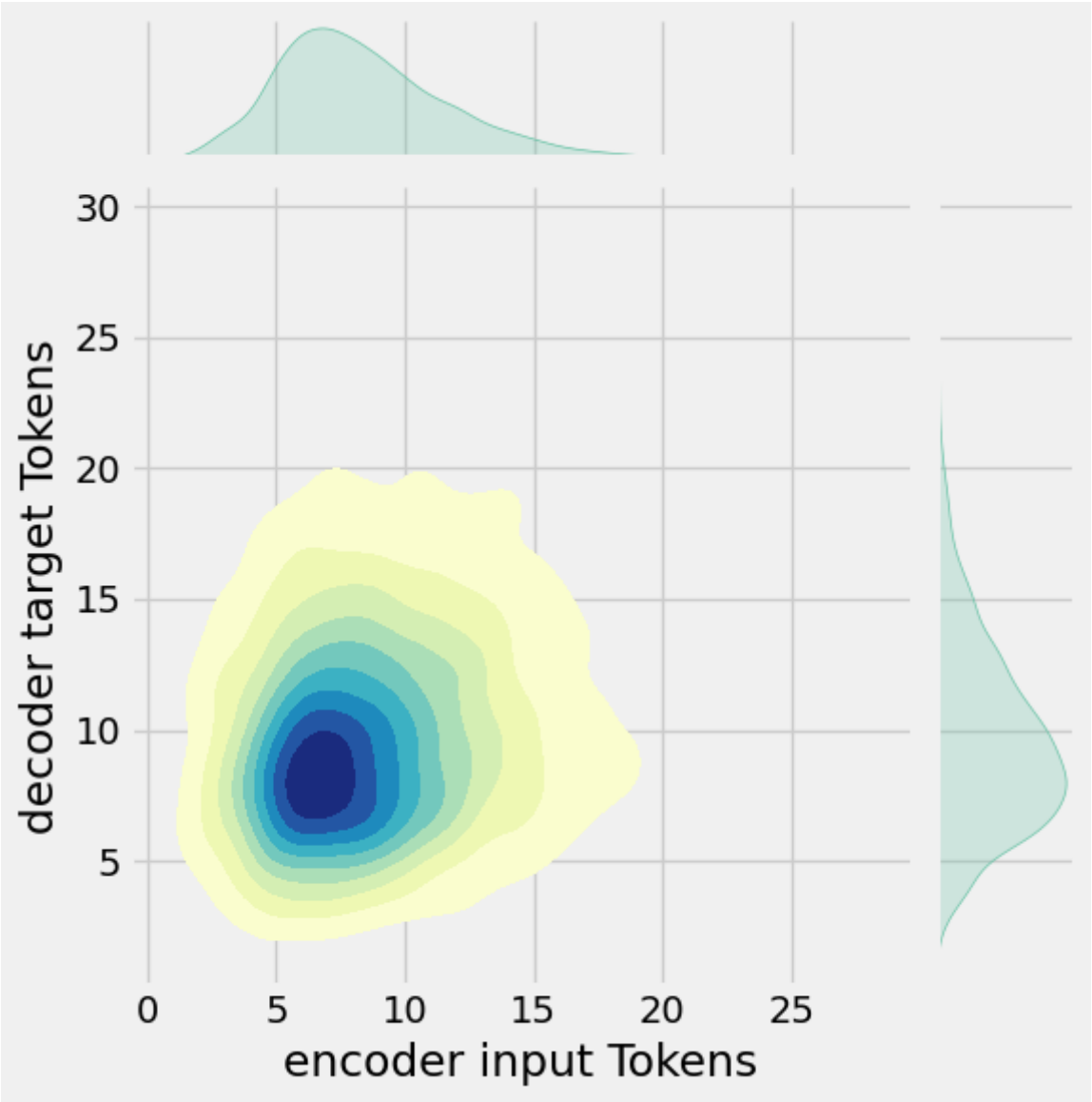
Now that we have cleaned the Dataset, Let us vizualize again

Re:Data Visualization

```
df['encoder input Tokens']=df['encoder_inputs'].apply(lambda
x:len(x.split()))
df['decoder input Tokens']=df['decoder_inputs'].apply(lambda
x:len(x.split()))
df['decoder target Tokens']=df['decoder_targets'].apply(lambda
```

```
x:len(x.split()))
plt.style.use('fivethirtyeight')
fig,ax=plt.subplots(nrows=1,ncols=3,figsize=(20,5))
sns.set_palette('Set2')
sns.histplot(x=df['encoder input Tokens'],data=df,kde=True,ax=ax[0])
sns.histplot(x=df['decoder input Tokens'],data=df,kde=True,ax=ax[1])
sns.histplot(x=df['decoder target Tokens'],data=df,kde=True,ax=ax[2])
sns.jointplot(x='encoder input Tokens',y='decoder target
Tokens',data=df,kind='kde',fill=True,cmap='YlGnBu')
plt.show()
```






```
print(f"After preprocessing: {' '.join(df[df['encoder input
tokens'].max()==df['encoder input tokens']]
['encoder_inputs'].values.tolist())}")
print(f"Max encoder input length: {df['encoder input tokens'].max()}")
print(f"Max decoder input length: {df['decoder input tokens'].max()}")
print(f"Max decoder target length: {df['decoder target tokens'].max()}")

df.drop(columns=['question','answer','encoder input tokens','decoder
```

```
input tokens','decoder target tokens'],axis=1,inplace=True)
params={
    "vocab_size":2500,
    "max_sequence_length":30,
    "learning_rate":0.008,
    "batch_size":149,
    "lstm_cells":256,
    "embedding_dim":256,
    "buffer_size":10000
}
learning_rate=params['learning_rate']
batch_size=params['batch_size']
embedding_dim=params['embedding_dim']
lstm_cells=params['lstm_cells']
vocab_size=params['vocab_size']
buffer_size=params['buffer_size']
max_sequence_length=params['max_sequence_length']
df.head(10)
```

```
After preprocessing: for example ,  if your birth date is january  1  2
,  1  9  8  7  ,  write  0  1  /  1  2  /  8  7  .
Max encoder input length: 27
Max decoder input length: 29
Max decoder target length: 28
```

	encoder_inputs	decoder_targets
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 ...
5	i ' ve been good . i ' m in school right now .	what school do you go to ?
6	what school do you go to ?	i go to pcc .
7	i go to pcc .	do you like it there ?
8	do you like it there ?	it ' s okay . it ' s a really big campus . <...
9	it ' s okay . it ' s a really big campus .	good luck with school .

Here is the Breakdown of the code

1. Print the longest encoder input sequence to the console.
2. Print the maximum encoder input length, decoder input length, and decoder target length to the console.
3. Drop the `question`, `answer`, `encoder input tokens`, `decoder input tokens`, and `decoder target tokens` columns from the `df` DataFrame.
4. Define a dictionary called `params` that contains the hyperparameters for the ChatBot model.
5. Define variables for the learning rate, batch size, embedding dimension, number of LSTM cells, vocabulary size, buffer size,and maximum sequence length.
6. Print the first 10 rows of the `df` DataFrame to the console.

Now that we know we haven't messed up our dataset let us proceed with Tokenization.

Tokenization

Now let us break down our text into Tokens.

```
vectorize_layer=TextVectorization(
    max_tokens=vocab_size,
    standardize=None,
    output_mode='int',
    output_sequence_length=max_sequence_length
)
vectorize_layer.adapt(df['encoder_inputs']+' '+df['decoder_targets']+'
<start> <end>')
vocab_size=len(vectorize_layer.get_vocabulary())
print(f'Vocab size: {len(vectorize_layer.get_vocabulary())}')
print(f'{vectorize_layer.get_vocabulary()[:12]}')
```

For Tokenization we will create a `TextVectorization` layer with the following parameters:

- `max_tokens`: The maximum number of tokens to consider when creating the vocabulary.
- `standardize`: A function to standardize the text before tokenization. This can be useful for removing punctuation, stop words, and other irrelevant tokens.
- `output_mode`: The output mode of the layer. This can be `int` (to output integer token indices), `binary` (to output a binary vector indicating whether or not each token is present in the text), or `count` (to output a vector of token counts).
- `output_sequence_length`: The maximum length of the output sequence. This can be used to pad or truncate sequences to a consistent length.

The `TextVectorization` layer is used to convert text tokens into integer indices. This is useful because it allows machine learning models to operate on text data.

The `TextVectorization` layer is used in the ChatBot model to convert the encoder and decoder inputs into integer indices. This allows the encoder and decoder models to process the text data and generate text responses.

Now we will need to prepare the `vectorize_layer` layer for training the ChatBot model. The `vectorize_layer` layer will be used to convert the encoder and decoder inputs and outputs to integer indices. This is necessary because machine learning models can only operate on numerical data.

To do so we will:

1. Adapt the `vectorize_layer` layer to the encoder inputs and decoder targets, plus the `<start>` and `<end>` tokens. This means that the layer will learn the vocabulary of the data and create a mapping from tokens to integer indices.
2. Get the size of the vocabulary and prints it to the console.
3. Print the first 12 tokens in the vocabulary to the console.

```
Vocab size: 2443
['', '[UNK]', '<end>', '.', '<start>', "'", 'i', '?', 'you', ',', 'the',
'to']
```

```
def sequences2ids(sequence):
    return vectorize_layer(sequence)

def ids2sequences(ids):
    decode=''
    if type(ids)==int:
        ids=[ids]
    for id in ids:
        decode+=vectorize_layer.get_vocabulary()[id]+' '
    return decode

x=sequences2ids(df['encoder_inputs'])
yd=sequences2ids(df['decoder_inputs'])
y=sequences2ids(df['decoder_targets'])
```

Code Breakdown

- `sequences2ids()`: This function converts a sequence of text tokens to a sequence of integer indices.
- `ids2sequences()`: This function converts a sequence of integer indices to a sequence of text tokens.

The `sequences2ids()` function uses the `vectorize_layer` layer to convert the sequence of text tokens to integer indices. This is so that the layer knows how to map each token in the vocabulary to an integer index.

The `ids2sequences()` function uses the `vectorize_layer.get_vocabulary()` method to get a list of all the tokens in the vocabulary. The function then iterates over the sequence of integer indices and replaces each index with the corresponding token in the vocabulary.

Let us do few Tests

```
print(f'Question sentence: hi , how are you ?')
print(f'Question to tokens: {sequences2ids("hi , how are you ?")[:10]}')
print(f'Encoder input shape: {x.shape}')
print(f'Decoder input shape: {yd.shape}')
print(f'Decoder target shape: {y.shape}')
```

```
Question sentence: hi , how are you ?
Question to tokens: [1971    9   45   24    8    7    0    0    0    0]
Encoder input shape: (3725, 30)
Decoder input shape: (3725, 30)
Decoder target shape: (3725, 30)
```

```
print(f'Encoder input: {x[0][:12]} ...')
print(f'Decoder input: {yd[0][:12]} ...')
print(f'Decoder target: {y[0][:12]} ...')
```

```
Encoder input: [1971    9   45   24    8  194    7    0    0    0    0
0] ...
Decoder input: [  4   6   5  38 646   3  45  41 563   7   2   0] ...
Decoder target: [  6   5  38 646   3  45  41 563   7   2   0   0] ...
```

The code prints the first 12 tokens of the encoder input, decoder input, and decoder target to the console. The decoder input is shifted by one time step of the target, which means that the output of the previous time step is used as the input to the current time step.

Now let us create a TensorFlow `Dataset` object and then prepares it for training and validation.

- Let `tf.data.Dataset.from_tensor_slices()` function creates a `Dataset` object from a tuple of NumPy arrays. The `(x, yd, y)` tuple contains the encoder input, decoder input, and decoder target tensors, respectively.
- `shuffle()` function shuffles the `Dataset` object. This helps to prevent the model from overfitting to the training data.
- Let `take()` and `skip()` functions split the `Dataset` object into two subsets: a training subset and a validation subset. The training subset contains 90% of the data, and the validation subset contains 10% of the data.
- Let `cache()` function caches the `Dataset` object in memory. This can improve the performance of the model, especially if the training data is large.
- Let `batch()` function batches the `Dataset` object. This means that the data will be fed to the model in batches of `batch_size` samples.
- Let `prefetch()` function prefetches the next batch of data while the model is training. This can improve the performance of the model by overlapping the data loading and training operations.
- Let `as_numpy_iterator()` function converts the `Dataset` object to a NumPy iterator. This allows the data to be iterated over in a NumPy-style fashion.
- Let `_=train_data_iterator.next()` line of code fetches the next batch of training data. This is done so that the training data iterator is ready to be used when the training loop begins.

```
data=tf.data.Dataset.from_tensor_slices((x,yd,y))
data=data.shuffle(buffer_size) train_data=data.take(int(.9*len(data)))
train_data=train_data.cache() train_data=train_data.shuffle(buffer_size)
train_data=train_data.batch(batch_size)
train_data=train_data.prefetch(tf.data.AUTOTUNE)
train_data_iterator=train_data.as_numpy_iterator()
val_data=data.skip(int(.9*len(data))).take(int(.1*len(data)))
val_data=val_data.batch(batch_size)
val_data=val_data.prefetch(tf.data.AUTOTUNE)
_=train_data_iterator.next()
```

The `Dataset` object is now ready to be used to train and validate the ChatBot model.

Let us verify a bit

```
print(f'Number of train batches: {len(train_data)}')
print(f'Number of training data: {len(train_data)*batch_size}')
```

```
print(f'Number of validation batches: {len(val_data)}')
print(f'Number of validation data: {len(val_data)*batch_size}')
print(f'Encoder Input shape (with batches): {_[0].shape}')
print(f'Decoder Input shape (with batches): {_[1].shape}')
print(f'Target Output shape (with batches): {_[2].shape}')
Number of train batches: 23
Number of training data: 3427
Number of validation batches: 3
Number of validation data: 447
Encoder Input shape (with batches): (149, 30)
Decoder Input shape (with batches): (149, 30)
Target Output shape (with batches): (149, 30)
```

With this we have successfully Preprocessed the Dataset

Build Models

Building Encoder

Let us define an encoder class for a ChatBot model. The encoder class inherits from the `tf.keras.models.Model` class, which means that it can be trained and used to make predictions like any other TensorFlow model.

The encoder class has the following attributes:

- `units`: The number of units in the LSTM layer.
- `embedding_dim`: The dimension of the word embeddings.
- `vocab_size`: The size of the vocabulary.

The encoder class also has the following layers:

- `embedding`: An embedding layer that converts the input tokens to word embeddings.
- `normalize`: A layer normalization layer that normalizes the output of the embedding layer.
- `lstm`: An LSTM layer that encodes the input word embeddings.

The encoder class has the following methods:

- `__init__()`: The constructor for the encoder class.
- `call()`: The forward pass method for the encoder class. This method takes the encoder inputs as input and returns the encoder outputs and hidden states.

The `encoder.call()` method is used to encode the input text sequence into a hidden state representation. The hidden state representation is then used by the decoder to generate the

output text sequence.

The `encoder.call(_[0])` line of code calls the `encoder.call()` method with the first batch of training data as input. This is done to ensure that the encoder is ready to be used when the training loop begins.

```
class Encoder(tf.keras.models.Model):
    def __init__(self,units,embedding_dim,vocab_size,*args,**kwargs) ->
None:
        super().__init__(*args,**kwargs)
        self.units=units
        self.vocab_size=vocab_size
        self.embedding_dim=embedding_dim
        self.embedding=Embedding(
            vocab_size,
            embedding_dim,
            name='encoder_embedding',
            mask_zero=True,
            embeddings_initializer=tf.keras.initializers.GlorotNormal()
        )
        self.normalize=LayerNormalization()
        self.lstm=LSTM(
            units,
            dropout=.4,
            return_state=True,
            return_sequences=True,
            name='encoder_lstm',
            kernel_initializer=tf.keras.initializers.GlorotNormal()
        )

    def call(self,encoder_inputs):
        self.inputs=encoder_inputs
        x=self.embedding(encoder_inputs)
        x=self.normalize(x)
        x=Dropout(.4)(x)
        encoder_outputs,encoder_state_h,encoder_state_c=self.lstm(x)
        self.outputs=[encoder_state_h,encoder_state_c]
        return encoder_state_h,encoder_state_c

encoder=Encoder(lstm_cells,embedding_dim,vocab_size,name='encoder')
encoder.call(_[0])
```

```
(<tf.Tensor: shape=(149, 256), dtype=float32, numpy=
 array([[ 0.16966951, -0.10419625, -0.12700348, ..., -0.12251794,
         0.10568858,  0.14841646],
```

```
        [ 0.08443093,  0.08849293, -0.09065959, ..., -0.00959182,
          0.10152507, -0.12077457],
        [ 0.03628462, -0.02653611, -0.11506603, ..., -0.14669597,
          0.10292757,  0.13625325],
        ...,
        [-0.14210635, -0.12942064, -0.03288083, ...,  0.0568463 ,
         -0.02598592, -0.22455114],
        [ 0.20819993,  0.01196991, -0.09635217, ..., -0.18782297,
          0.10233591,  0.20114912],
        [ 0.1164271 , -0.07769038, -0.06414707, ..., -0.06539135,
         -0.05518465,  0.25142196]], dtype=float32)>,
<tf.Tensor: shape=(149, 256), dtype=float32, numpy=
array([[ 0.34589   , -0.30134732, -0.43572   , ..., -0.3102559 ,
         0.34630865,  0.2613009 ],
       [ 0.14154069,  0.17045322, -0.17749965, ..., -0.02712595,
         0.17292541, -0.2922624 ],
       [ 0.07106856, -0.0739173 , -0.3641197 , ..., -0.3794833 ,
         0.36470377,  0.23766585],
       ...,
       [-0.2582597 , -0.25323495, -0.06649272, ...,  0.16527973,
        -0.04292646, -0.58768904],
       [ 0.43155715,  0.03135502, -0.33463806, ..., -0.47625306,
         0.33486888,  0.35035062],
       [ 0.23173636, -0.20141824, -0.22034441, ..., -0.16035017,
        -0.17478186,  0.48899865]], dtype=float32)>)
```

Building Decoder

Now that we have Encoder, Let us create a decoder

let us defines a decoder class for a ChatBot model. The decoder class similar to encoder inherits from the `tf.keras.models.Model` class, which means that it can be trained and used to make predictions like any other TensorFlow model.

The decoder class has the following attributes:

- `units`: The number of units in the LSTM layer.
- `embedding_dim`: The dimension of the word embeddings.
- `vocab_size`: The size of the vocabulary.

The decoder class also has the following layers:

- `embedding`: An embedding layer that converts the decoder inputs to word embeddings.
- `normalize`: A layer normalization layer that normalizes the output of the embedding layer.

- `lstm`: An LSTM layer that decodes the input word embeddings.
- `fc`: A dense layer that generates the output tokens.

The decoder class has the following methods:

- `__init__()`: The constructor for the decoder class.
- `call()`: The forward pass method for the decoder class. This method takes the decoder inputs and encoder states as input and returns the decoder outputs.

The `decoder.call()` method is used to decode the encoder hidden state representation into a sequence of output tokens.

The `decoder(_[1][:1],encoder(_[0][:1]))` line of code calls the `decoder.call()` method with the first batch of training data as input. This is done to ensure that the decoder is ready to be used when the training loop begins.

```
class Decoder(tf.keras.models.Model):

def __init__(self,units,embedding_dim,vocab_size,*args,**kwargs) →
None:

super().__init__(*args,**kwargs)

self.units=units

self.embedding_dim=embedding_dim

self.vocab_size=vocab_size

self.embedding=Embedding(

vocab_size,

embedding_dim,

name='decoder_embedding',

mask_zero=True,

embeddings_initializer=tf.keras.initializers.HeNormal()

)

self.normalize=LayerNormalization()
```

```
self.lstm=LSTM(

units,

dropout=.4,

return_state=True,

return_sequences=True,

name='decoder_lstm',

kernel_initializer=tf.keras.initializers.HeNormal()

)

self.fc=Dense(

vocab_size,

activation='softmax',

name='decoder_dense',

kernel_initializer=tf.keras.initializers.HeNormal()

)

def call(self,decoder_inputs,encoder_states):

x=self.embedding(decoder_inputs)

x=self.normalize(x)

x=Dropout(.4)(x)

x,decoder_state_h,decoder_state_c=self.lstm(x,initial_state=encoder_stat
es)

x=self.normalize(x)

x=Dropout(.4)(x)

return self.fc(x)
```

```
decoder=Decoder(lstm_cells,embedding_dim,vocab_size,name='decoder')

decoder(_[1][:1],encoder(_[0][:1]))
```

```
<tf.Tensor: shape=(1, 30, 2443), dtype=float32, numpy=
array([[[3.4059247e-04, 5.7348556e-05, 2.1294907e-05, ...,
         7.2067953e-05, 1.5453645e-03, 2.3599296e-04],
        [1.4662130e-03, 8.0250365e-06, 5.4062020e-05, ...,
         1.9187471e-05, 9.7244098e-05, 7.6433855e-05],
        [9.6929165e-05, 2.7441782e-05, 1.3761305e-03, ...,
         3.6009602e-05, 1.5537882e-04, 1.8397317e-04],
        ...,
        [1.9002777e-03, 6.9266016e-04, 1.4346189e-04, ...,
         1.9552530e-04, 1.7106640e-05, 1.0252406e-04],
        [1.9002777e-03, 6.9266016e-04, 1.4346189e-04, ...,
         1.9552530e-04, 1.7106640e-05, 1.0252406e-04],
        [1.9002777e-03, 6.9266016e-04, 1.4346189e-04, ...,
         1.9552530e-04, 1.7106640e-05, 1.0252406e-04]]], dtype=float32)>
```

Building Training Model

Similarly defines a `ChatBotTrainer` class for training a ChatBot model. The `ChatBotTrainer` class inherits from the `tf.keras.models.Model` class.

The `ChatBotTrainer` class has the following attributes:

- `encoder`: The encoder model.
- `decoder`: The decoder model.

The `ChatBotTrainer` class also has the following methods:

- `__init__()`: The constructor for the `ChatBotTrainer` class.
- `loss_fn()`: The loss function for the ChatBot model.
- `accuracy_fn()`: The accuracy function for the ChatBot model.
- `call()`: The forward pass method for the ChatBot model.
- `train_step()`: The training step for the ChatBot model.
- `test_step()`: The testing step for the ChatBot model.

The `ChatBotTrainer` class is responsible for training the ChatBot model and making predictions.

```
class ChatBotTrainer(tf.keras.models.Model):
    def __init__(self,encoder,decoder,*args,**kwargs):
        super().__init__(*args,**kwargs)
        self.encoder=encoder
        self.decoder=decoder

    def loss_fn(self,y_true,y_pred):
        loss=self.loss(y_true,y_pred)
        mask=tf.math.logical_not(tf.math.equal(y_true,0))
        mask=tf.cast(mask,dtype=loss.dtype)
        loss*=mask
        return tf.reduce_mean(loss)

    def accuracy_fn(self,y_true,y_pred):
        pred_values = tf.cast(tf.argmax(y_pred, axis=-1), dtype='int64')
        correct = tf.cast(tf.equal(y_true, pred_values),
dtype='float64')
        mask = tf.cast(tf.greater(y_true, 0), dtype='float64')
        n_correct = tf.keras.backend.sum(mask * correct)
        n_total = tf.keras.backend.sum(mask)
        return n_correct / n_total

    def call(self,inputs):
        encoder_inputs,decoder_inputs=inputs
        encoder_states=self.encoder(encoder_inputs)
        return self.decoder(decoder_inputs,encoder_states)

    def train_step(self,batch):
        encoder_inputs,decoder_inputs,y=batch
        with tf.GradientTape() as tape:
            encoder_states=self.encoder(encoder_inputs,training=True)

y_pred=self.decoder(decoder_inputs,encoder_states,training=True)
            loss=self.loss_fn(y,y_pred)
            acc=self.accuracy_fn(y,y_pred)


variables=self.encoder.trainable_variables+self.decoder.trainable_variab
les
        grads=tape.gradient(loss,variables)
        self.optimizer.apply_gradients(zip(grads,variables))
        metrics={'loss':loss,'accuracy':acc}
        return metrics

    def test_step(self,batch):
        encoder_inputs,decoder_inputs,y=batch
```

```
        encoder_states=self.encoder(encoder_inputs,training=True)
        y_pred=self.decoder(decoder_inputs,encoder_states,training=True)
        loss=self.loss_fn(y,y_pred)
        acc=self.accuracy_fn(y,y_pred)
        metrics={'loss':loss,'accuracy':acc}
        return metrics
```

Let us do a small test

```
model=ChatBotTrainer(encoder,decoder,name='chatbot_trainer')
model.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
    weighted_metrics=['loss','accuracy']
)
model(_[:2])
```

```
<tf.Tensor: shape=(149, 30, 2443), dtype=float32, numpy=
array([[[3.40592262e-04, 5.73484940e-05, 2.12948853e-05, ...,
         7.20679745e-05, 1.54536311e-03, 2.35993255e-04],
        [1.46621116e-03, 8.02504110e-06, 5.40619949e-05, ...,
         1.91874733e-05, 9.72440175e-05, 7.64339056e-05],
        [9.69291723e-05, 2.74417835e-05, 1.37613132e-03, ...,
         3.60095728e-05, 1.55378671e-04, 1.83973272e-04],
        ...,
        [1.90027885e-03, 6.92659756e-04, 1.43461803e-04, ...,
         1.95525470e-04, 1.71066222e-05, 1.02524005e-04],
        [1.90027885e-03, 6.92659756e-04, 1.43461803e-04, ...,
         1.95525470e-04, 1.71066222e-05, 1.02524005e-04],
        [1.90027885e-03, 6.92659756e-04, 1.43461803e-04, ...,
         1.95525470e-04, 1.71066222e-05, 1.02524005e-04]],

       [[9.24730921e-05, 3.46553512e-04, 2.07866033e-05, ...,
         3.65934626e-04, 7.63039337e-04, 5.52638434e-04],
        [8.46863186e-05, 3.65541164e-05, 2.54740953e-05, ...,
         7.12379551e-05, 3.62201303e-04, 4.16714087e-04],
        [2.30146630e-04, 3.91469621e-06, 2.72463716e-04, ...,
         9.26126595e-05, 1.03836363e-04, 1.40792166e-04],
        ...,
        [6.84961735e-04, 9.07644513e-04, 2.86691647e-04, ...,
         3.87946144e-04, 6.09236558e-05, 1.12995331e-05],
        [6.84961735e-04, 9.07644513e-04, 2.86691647e-04, ...,
         3.87946144e-04, 6.09236558e-05, 1.12995331e-05],
        [6.84961735e-04, 9.07644513e-04, 2.86691647e-04, ...,
         3.87946144e-04, 6.09236558e-05, 1.12995322e-05]],
```

```
[[1.19036995e-03, 8.10516722e-05, 2.42324077e-05, ...,
  4.99442758e-05, 6.67208573e-04, 9.55566764e-04],
 [1.53046989e-04, 9.76863957e-05, 4.96972689e-06, ...,
  3.24743196e-05, 2.12563842e-04, 1.18708890e-03],
 [9.40205529e-04, 1.80782794e-04, 7.26205144e-06, ...,
  1.96355060e-04, 8.16940737e-05, 1.38416886e-03],
 ...,
 [3.52622545e-03, 1.26781175e-03, 1.02695449e-04, ...,
  2.35450850e-03, 3.25187625e-06, 9.46984728e-05],
 [3.52622545e-03, 1.26781175e-03, 1.02695449e-04, ...,
  2.35450850e-03, 3.25187625e-06, 9.46984728e-05],
 [3.52622545e-03, 1.26781175e-03, 1.02695449e-04, ...,
  2.35450850e-03, 3.25187625e-06, 9.46984728e-05]],

...,

[[9.03617911e-05, 1.57651404e-04, 1.02747028e-04, ...,
  2.20922651e-04, 3.61504179e-04, 2.32456136e-03],
 [1.55469708e-04, 1.53608169e-04, 1.14945491e-04, ...,
  1.88878359e-04, 5.11967926e-04, 5.13108505e-04],
 [8.27641197e-05, 2.83437112e-05, 6.29429938e-04, ...,
  2.15980137e-04, 3.02832137e-04, 1.77760507e-04],
 ...,
 [2.41102395e-03, 1.29279669e-03, 9.11735406e-05, ...,
  4.06600971e-04, 7.58682154e-06, 6.05909081e-05],
 [2.41102395e-03, 1.29279669e-03, 9.11735406e-05, ...,
  4.06600971e-04, 7.58682154e-06, 6.05909081e-05],
 [2.41102395e-03, 1.29279669e-03, 9.11735406e-05, ...,
  4.06600971e-04, 7.58682154e-06, 6.05909081e-05]],

[[3.99837241e-04, 2.36026899e-05, 6.89777007e-05, ...,
  5.94239136e-05, 4.32556757e-04, 4.60232928e-04],
 [3.88111075e-04, 8.31133584e-05, 1.11861555e-04, ...,
  3.03280340e-05, 2.54765386e-04, 2.82170397e-04],
 [2.12516752e-03, 7.19837190e-05, 1.88700986e-04, ...,
  1.86366087e-04, 7.02239413e-05, 2.54370330e-04],
 ...,
 [4.56329063e-03, 2.23812275e-03, 2.37343236e-04, ...,
  2.64523784e-04, 4.05454011e-05, 1.55662783e-04],
 [4.56329063e-03, 2.23812275e-03, 2.37343236e-04, ...,
  2.64523784e-04, 4.05454011e-05, 1.55662783e-04],
 [4.56329063e-03, 2.23812275e-03, 2.37343236e-04, ...,
  2.64523784e-04, 4.05454011e-05, 1.55662783e-04]],

[[3.24600202e-04, 9.31067043e-05, 4.60048941e-05, ...,
```

```
          6.66230699e-05, 5.76460850e-04, 1.52416309e-04],
         [7.51478728e-05, 7.63997741e-05, 2.09082973e-05, ...,
          2.55555002e-04, 2.28998848e-04, 4.37303359e-04],
         [1.03114333e-04, 1.55743372e-04, 9.97955431e-06, ...,
          1.12485175e-03, 4.80950950e-03, 6.83143327e-04],
         ...,
         [5.20280097e-03, 3.23211338e-04, 2.47709468e-05, ...,
          3.07609705e-04, 6.09844255e-06, 8.61325825e-05],
         [5.20280097e-03, 3.23211338e-04, 2.47709468e-05, ...,
          3.07609705e-04, 6.09844255e-06, 8.61325825e-05],
         [5.20280097e-03, 3.23211338e-04, 2.47709468e-05, ...,
          3.07609705e-04, 6.09844255e-06, 8.61325825e-05]]],
dtype=float32)>
```

Train Model

This is a simple code which trains the ChatBot model for 100 epochs using the `train_data` and `val_data` datasets. It also uses two callbacks:

- `tf.keras.callbacks.TensorBoard(log_dir='logs')`: This callback logs training metrics to TensorBoard, which is a visualization tool for TensorFlow.
- `tf.keras.callbacks.ModelCheckpoint('ckpt',verbose=1,save_best_only=True)`: This callback saves the model parameters to the `ckpt` directory after each epoch. The model parameters with the best validation accuracy are saved.

The `history` variable contains the training history, which includes the loss and accuracy metrics for the training and validation datasets for each epoch.

```
history=model.fit(
    train_data,
    epochs=100,
    validation_data=val_data,
    callbacks=[
        tf.keras.callbacks.TensorBoard(log_dir='logs'),

tf.keras.callbacks.ModelCheckpoint('ckpt',verbose=1,save_best_only=True)
    ]
)
```

```
Epoch 1/100
23/23 [==============================] - ETA: 0s - loss: 1.6604 -
accuracy: 0.2179
Epoch 1: val_loss improved from inf to 1.34060, saving model to ckpt
```

```
23/23 [==============================] - 81s 3s/step - loss: 1.6474 -
accuracy: 0.2204 - val_loss: 1.3406 - val_accuracy: 0.2955
Epoch 2/100
23/23 [==============================] - ETA: 0s - loss: 1.2330 -
accuracy: 0.3075
Epoch 2: val_loss improved from 1.34060 to 1.02251, saving model to ckpt
23/23 [==============================] - 58s 3s/step - loss: 1.2308 -
accuracy: 0.3083 - val_loss: 1.0225 - val_accuracy: 0.3318
Epoch 3/100
23/23 [==============================] - ETA: 0s - loss: 1.1021 -
accuracy: 0.3360
Epoch 3: val_loss did not improve from 1.02251
23/23 [==============================] - 39s 2s/step - loss: 1.1018 -
accuracy: 0.3359 - val_loss: 1.0243 - val_accuracy: 0.3484
Epoch 4/100
23/23 [==============================] - ETA: 0s - loss: 1.0201 -
accuracy: 0.3545
Epoch 4: val_loss improved from 1.02251 to 1.01495, saving model to ckpt
23/23 [==============================] - 60s 3s/step - loss: 1.0208 -
accuracy: 0.3546 - val_loss: 1.0150 - val_accuracy: 0.3699
Epoch 5/100
23/23 [==============================] - ETA: 0s - loss: 0.9669 -
accuracy: 0.3676
Epoch 5: val_loss improved from 1.01495 to 0.99426, saving model to ckpt
23/23 [==============================] - 61s 3s/step - loss: 0.9738 -
accuracy: 0.3658 - val_loss: 0.9943 - val_accuracy: 0.3827
Epoch 6/100
23/23 [==============================] - ETA: 0s - loss: 0.9170 -
accuracy: 0.3807
Epoch 6: val_loss improved from 0.99426 to 0.90623, saving model to ckpt
23/23 [==============================] - 60s 3s/step - loss: 0.9186 -
accuracy: 0.3798 - val_loss: 0.9062 - val_accuracy: 0.3894
Epoch 7/100
23/23 [==============================] - ETA: 0s - loss: 0.8742 -
accuracy: 0.3926
Epoch 7: val_loss improved from 0.90623 to 0.87546, saving model to ckpt
23/23 [==============================] - 59s 3s/step - loss: 0.8731 -
accuracy: 0.3928 - val_loss: 0.8755 - val_accuracy: 0.4005
```

```
Epoch 8/100
23/23 [==============================] - ETA: 0s - loss: 0.8449 -
accuracy: 0.4010
Epoch 8: val_loss improved from 0.87546 to 0.76850, saving model to ckpt
23/23 [==============================] - 60s 3s/step - loss: 0.8494 -
accuracy: 0.3997 - val_loss: 0.7685 - val_accuracy: 0.4242
Epoch 9/100
23/23 [==============================] - ETA: 0s - loss: 0.8135 -
accuracy: 0.4098
Epoch 9: val_loss did not improve from 0.76850
23/23 [==============================] - 38s 2s/step - loss: 0.8118 -
accuracy: 0.4093 - val_loss: 0.8006 - val_accuracy: 0.4085
Epoch 10/100
23/23 [==============================] - ETA: 0s - loss: 0.7875 -
accuracy: 0.4230
Epoch 10: val_loss did not improve from 0.76850
23/23 [==============================] - 37s 2s/step - loss: 0.7910 -
accuracy: 0.4221 - val_loss: 0.9075 - val_accuracy: 0.4209
Epoch 11/100
23/23 [==============================] - ETA: 0s - loss: 0.7666 -
accuracy: 0.4277

....
...
..
.
..
...
....

23/23 [==============================] - ETA: 0s - loss: 0.3511 -
accuracy: 0.6686
Epoch 91: val_loss did not improve from 0.32423
23/23 [==============================] - 38s 2s/step - loss: 0.3526 -
accuracy: 0.6685 - val_loss: 0.4857 - val_accuracy: 0.6494
Epoch 92/100
23/23 [==============================] - ETA: 0s - loss: 0.3536 -
accuracy: 0.6670
Epoch 92: val_loss did not improve from 0.32423
23/23 [==============================] - 38s 2s/step - loss: 0.3554 -
```

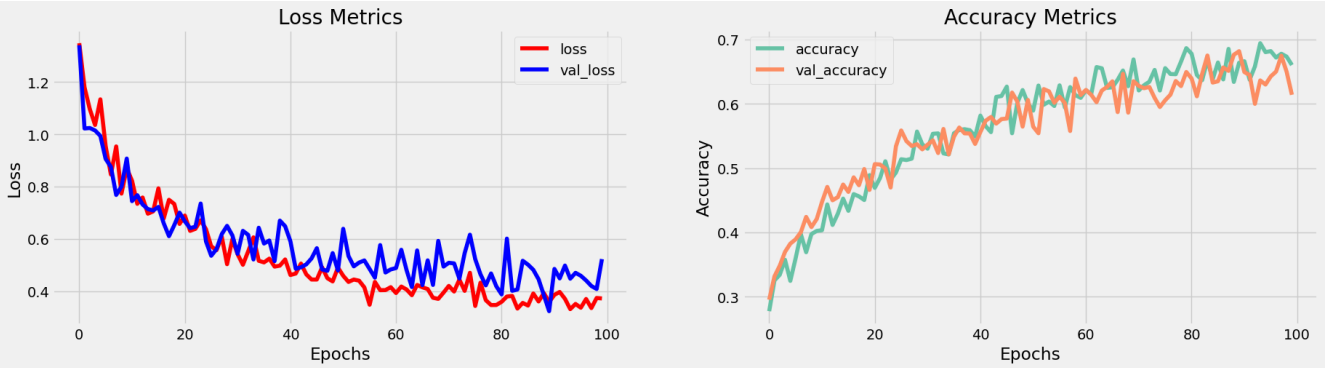
```
accuracy: 0.6658 - val_loss: 0.4494 - val_accuracy: 0.6447
Epoch 93/100
23/23 [==============================] - ETA: 0s - loss: 0.3510 -
accuracy: 0.6716
Epoch 93: val_loss did not improve from 0.32423
23/23 [==============================] - 37s 2s/step - loss: 0.3519 -
accuracy: 0.6711 - val_loss: 0.4985 - val_accuracy: 0.6000
Epoch 94/100
23/23 [==============================] - ETA: 0s - loss: 0.3462 -
accuracy: 0.6722
Epoch 94: val_loss did not improve from 0.32423
23/23 [==============================] - 38s 2s/step - loss: 0.3456 -
accuracy: 0.6731 - val_loss: 0.4485 - val_accuracy: 0.6364
Epoch 95/100
23/23 [==============================] - ETA: 0s - loss: 0.3488 -
accuracy: 0.6710
Epoch 95: val_loss did not improve from 0.32423
23/23 [==============================] - 37s 2s/step - loss: 0.3489 -
accuracy: 0.6714 - val_loss: 0.4708 - val_accuracy: 0.6300
Epoch 96/100
23/23 [==============================] - ETA: 0s - loss: 0.3418 -
accuracy: 0.6744
Epoch 96: val_loss did not improve from 0.32423
23/23 [==============================] - 38s 2s/step - loss: 0.3416 -
accuracy: 0.6747 - val_loss: 0.4596 - val_accuracy: 0.6427
Epoch 97/100
23/23 [==============================] - ETA: 0s - loss: 0.3410 -
accuracy: 0.6786
Epoch 97: val_loss did not improve from 0.32423
23/23 [==============================] - 38s 2s/step - loss: 0.3423 -
accuracy: 0.6783 - val_loss: 0.4417 - val_accuracy: 0.6501
Epoch 98/100
23/23 [==============================] - ETA: 0s - loss: 0.3426 -
accuracy: 0.6741
Epoch 98: val_loss did not improve from 0.32423
23/23 [==============================] - 37s 2s/step - loss: 0.3423 -
accuracy: 0.6743 - val_loss: 0.4205 - val_accuracy: 0.6748
Epoch 99/100
23/23 [==============================] - ETA: 0s - loss: 0.3444 -
accuracy: 0.6741
```

```
Epoch 99: val_loss did not improve from 0.32423
23/23 [==============================] - 37s 2s/step - loss: 0.3456 -
accuracy: 0.6741 - val_loss: 0.4090 - val_accuracy: 0.6499
Epoch 100/100
23/23 [==============================] - ETA: 0s - loss: 0.3451 -
accuracy: 0.6729
Epoch 100: val_loss did not improve from 0.32423
23/23 [==============================] - 37s 2s/step - loss: 0.3463 -
accuracy: 0.6724 - val_loss: 0.5233 - val_accuracy: 0.6141
```

Visualize Metrics

Let us visualize the model

```
fig,ax=plt.subplots(nrows=1,ncols=2,figsize=(20,5))

ax[0].plot(history.history['loss'],label='loss',c='red')

ax[0].plot(history.history['val_loss'],label='val_loss',c = 'blue')

ax[0].set_xlabel('Epochs')

ax[1].set_xlabel('Epochs')

ax[0].set_ylabel('Loss')

ax[1].set_ylabel('Accuracy')

ax[0].set_title('Loss Metrics')

ax[1].set_title('Accuracy Metrics')

ax[1].plot(history.history['accuracy'],label='accuracy')

ax[1].plot(history.history['val_accuracy'],label='val_accuracy')

ax[0].legend()

ax[1].legend()

plt.show()
```



The above graph shows that the training loss and accuracy metrics improve over time, while the validation loss and accuracy metrics plateau after a certain number of epochs. This suggests that the ChatBot model is overfitting to the training data.

Here are some key observations about the plot:

- The training loss metric decreases steadily over time, while the validation loss metric plateaus after about 20 epochs. This suggests that the ChatBot model is learning to fit the training data well, but it is not generalizing well to new data.
- The training accuracy metric increases steadily over time, while the validation accuracy metric plateaus after about 20 epochs. This also suggests that the ChatBot model is overfitting to the training data.

To address the overfitting problem, the following steps can be taken:

- Reduce the number of epochs trained.
- Use a smaller learning rate.
- Add regularization to the model, such as L1 or L2 regularization.
- Use dropout during training.
- Collect more training data.

Overall, the plot suggests that the ChatBot model is capable of learning to fit the training data well, but it is important to take steps to address the overfitting problem before deploying the model to production.

Save the Model

```
model.load_weights('ckpt')
model.save('models',save_format='tf')
for idx,i in enumerate(model.layers):
    print('Encoder layers:' if idx==0 else 'Decoder layers: ')
    for j in i.layers:
```

```
        print(j)
    print('--------------------')
```

```
Encoder layers:
<keras.src.layers.core.embedding.Embedding object at 0x79379dd34040>
<keras.src.layers.normalization.layer_normalization.LayerNormalization
object at 0x79379dd349a0>
<keras.src.layers.rnn.lstm.LSTM object at 0x79379dd3a0b0>
--------------------
Decoder layers: <keras.src.layers.core.embedding.Embedding object at
0x7937945c10c0>
<keras.src.layers.normalization.layer_normalization.LayerNormalization
object at 0x7937945c1cc0>
<keras.src.layers.rnn.lstm.LSTM object at 0x7937945c1c90>
<keras.src.layers.core.dense.Dense object at 0x7937945c2860>
--------------------
```

Create Inference Model

Let Us now create a interface for out chatbot

```
class ChatBot(tf.keras.models.Model):
    def __init__(self,base_encoder,base_decoder,*args,**kwargs):
        super().__init__(*args,**kwargs)

self.encoder,self.decoder=self.build_inference_model(base_encoder,base_d
ecoder)

    def build_inference_model(self,base_encoder,base_decoder):
        encoder_inputs=tf.keras.Input(shape=(None,))
        x=base_encoder.layers[0](encoder_inputs)
        x=base_encoder.layers[1](x)
        x,encoder_state_h,encoder_state_c=base_encoder.layers[2](x)
        encoder=tf.keras.models.Model(inputs=encoder_inputs,outputs=
[encoder_state_h,encoder_state_c],name='chatbot_encoder')

        decoder_input_state_h=tf.keras.Input(shape=(lstm_cells,))
        decoder_input_state_c=tf.keras.Input(shape=(lstm_cells,))
        decoder_inputs=tf.keras.Input(shape=(None,))
        x=base_decoder.layers[0](decoder_inputs)
        x=base_encoder.layers[1](x)
        x,decoder_state_h,decoder_state_c=base_decoder.layers[2]
```

```
(x,initial_state=[decoder_input_state_h,decoder_input_state_c])
        decoder_outputs=base_decoder.layers[-1](x)
        decoder=tf.keras.models.Model(
            inputs=[decoder_inputs,
[decoder_input_state_h,decoder_input_state_c]],
            outputs=[decoder_outputs,
[decoder_state_h,decoder_state_c]],name='chatbot_decoder'
        )
        return encoder,decoder

    def summary(self):
        self.encoder.summary()
        self.decoder.summary()

    def softmax(self,z):
        return np.exp(z)/sum(np.exp(z))

    def sample(self,conditional_probability,temperature=0.5):
        conditional_probability =
np.asarray(conditional_probability).astype("float64")
        conditional_probability = np.log(conditional_probability) /
temperature
        reweighted_conditional_probability =
self.softmax(conditional_probability)
        probas = np.random.multinomial(1,
reweighted_conditional_probability, 1)
        return np.argmax(probas)

    def preprocess(self,text):
        text=clean_text(text)
        seq=np.zeros((1,max_sequence_length),dtype=np.int32)
        for i,word in enumerate(text.split()):
            seq[:,i]=sequences2ids(word).numpy()[0]
        return seq

    def postprocess(self,text):
        text=re.sub(' - ','-',text.lower())
        text=re.sub(' [.] ','. ',text)
        text=re.sub(' [1] ','1',text)
        text=re.sub(' [2] ','2',text)
        text=re.sub(' [3] ','3',text)
        text=re.sub(' [4] ','4',text)
        text=re.sub(' [5] ','5',text)
        text=re.sub(' [6] ','6',text)
        text=re.sub(' [7] ','7',text)
        text=re.sub(' [8] ','8',text)
```

```
        text=re.sub(' [9] ','9',text)
        text=re.sub(' [0] ','0',text)
        text=re.sub(' [,] ',', ',text)
        text=re.sub(' [?] ','? ',text)
        text=re.sub(' [!] ','! ',text)
        text=re.sub(' [$] ','$ ',text)
        text=re.sub(' [&] ','& ',text)
        text=re.sub(' [/] ','/ ',text)
        text=re.sub(' [:] ',': ',text)
        text=re.sub(' [;] ','; ',text)
        text=re.sub(' [*] ','* ',text)
        text=re.sub(' [\'] ','\'',text)
        text=re.sub(' [\"] ','\"',text)
        return text

    def call(self,text,config=None):
        input_seq=self.preprocess(text)
        states=self.encoder(input_seq,training=False)
        target_seq=np.zeros((1,1))
        target_seq[:,:]=sequences2ids(['<start>']).numpy()[0][0]
        stop_condition=False
        decoded=[]
        while not stop_condition:

decoder_outputs,new_states=self.decoder([target_seq,states],training=Fal
se)
#
index=tf.argmax(decoder_outputs[:,-1,:],axis=-1).numpy().item()
            index=self.sample(decoder_outputs[0,0,:]).item()
            word=ids2sequences([index])
            if word=='<end> ' or len(decoded)≥max_sequence_length:
                stop_condition=True
            else:
                decoded.append(index)
                target_seq=np.zeros((1,1))
                target_seq[:,:]=index
                states=new_states
        return self.postprocess(ids2sequences(decoded))

chatbot=ChatBot(model.encoder,model.decoder,name='chatbot')
chatbot.summary()
```

```
Model: "chatbot_encoder"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
```

```
=================================================================
 input_1 (InputLayer)        [(None, None)]            0

 encoder_embedding (Embeddin  (None, None, 256)        625408
 g)

 layer_normalization (LayerN  (None, None, 256)        512
 ormalization)

 encoder_lstm (LSTM)         [(None, None, 256),       525312
                              (None, 256),
                              (None, 256)]

=================================================================
Total params: 1,151,232
Trainable params: 1,151,232
Non-trainable params: 0
_________________________________________________________________
Model: "chatbot_decoder"
__________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #
Connected to
==========================================================================================
 input_4 (InputLayer)           [(None, None)]       0           []

 decoder_embedding (Embedding)  (None, None, 256)    625408
['input_4[0][0]']

 layer_normalization (LayerNorm  (None, None, 256)   512
['decoder_embedding[0][0]']
 alization)

 input_2 (InputLayer)           [(None, 256)]        0           []

 input_3 (InputLayer)           [(None, 256)]        0           []

 decoder_lstm (LSTM)            [(None, None, 256),  525312
['layer_normalization[1][0]',
```

```
                                 (None, 256),
'input_2[0][0]',
                                 (None, 256)]
'input_3[0][0]']

 decoder_dense (Dense)          (None, None, 2443)   627851
['decoder_lstm[0][0]']

==================================================================================
==========================
Total params: 1,779,083
Trainable params: 1,779,083
Non-trainable params: 0
__________________________________________________________________________________
__________________________
```

Since this is a simple interface we will not go into details but here is a simple code breakdown.

The `ChatBot` class has the following attributes:

- `encoder`: The encoder model.
- `decoder`: The decoder model.

The `ChatBot` class also has the following methods:

- `build_inference_model()`: This method builds the inference model for the ChatBot. The inference model is used to make predictions on new data.
- `summary()`: This method prints a summary of the ChatBot model.
- `softmax()`: This method calculates the softmax of a given vector.
- `sample()`: This method samples a word from a given probability distribution.
- `preprocess()`: This method preprocesses a text input for the ChatBot.
- `postprocess()`: This method postprocesses a text output from the ChatBot.
- `call()`: This method makes a prediction with the ChatBot model.

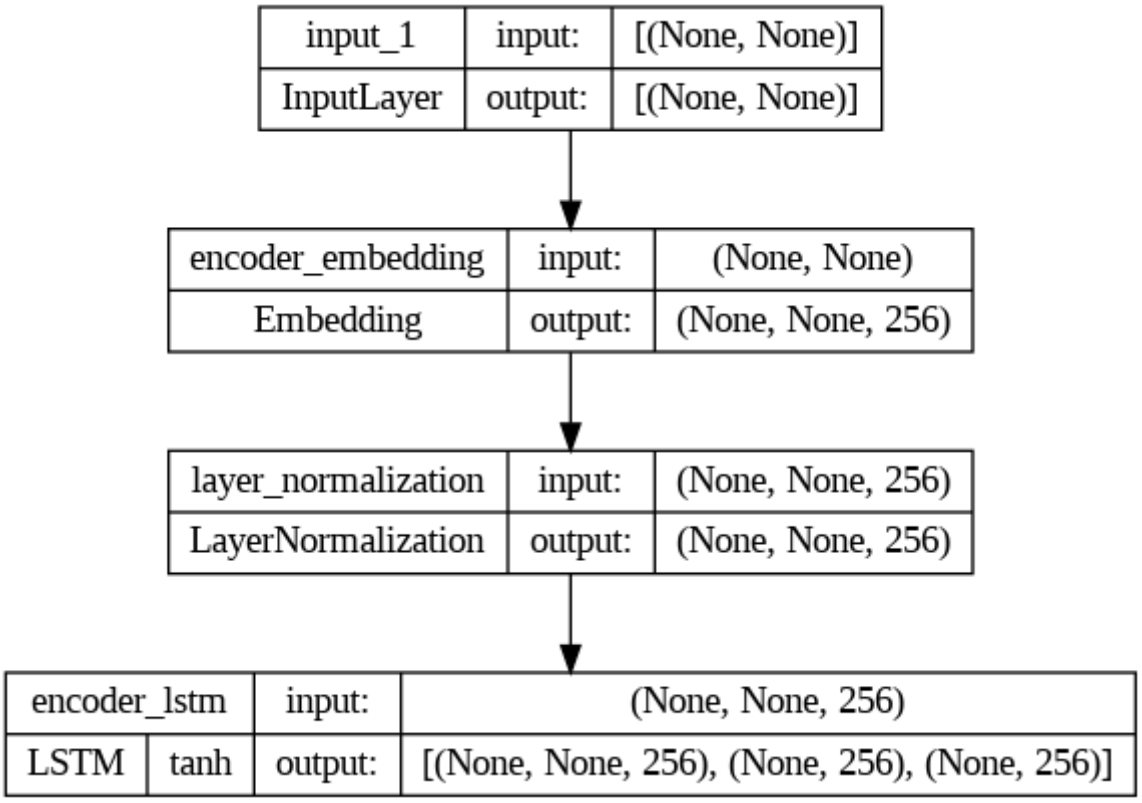
The `ChatBot.call()` method takes a text input as input and returns a text output. The text output is the best guess of the ChatBot model for what the next word in the conversation should be.

The `ChatBot` class is used to create a ChatBot model that can be used to generate text. The ChatBot model can be used to create chatbots, dialogue systems, and other text generation

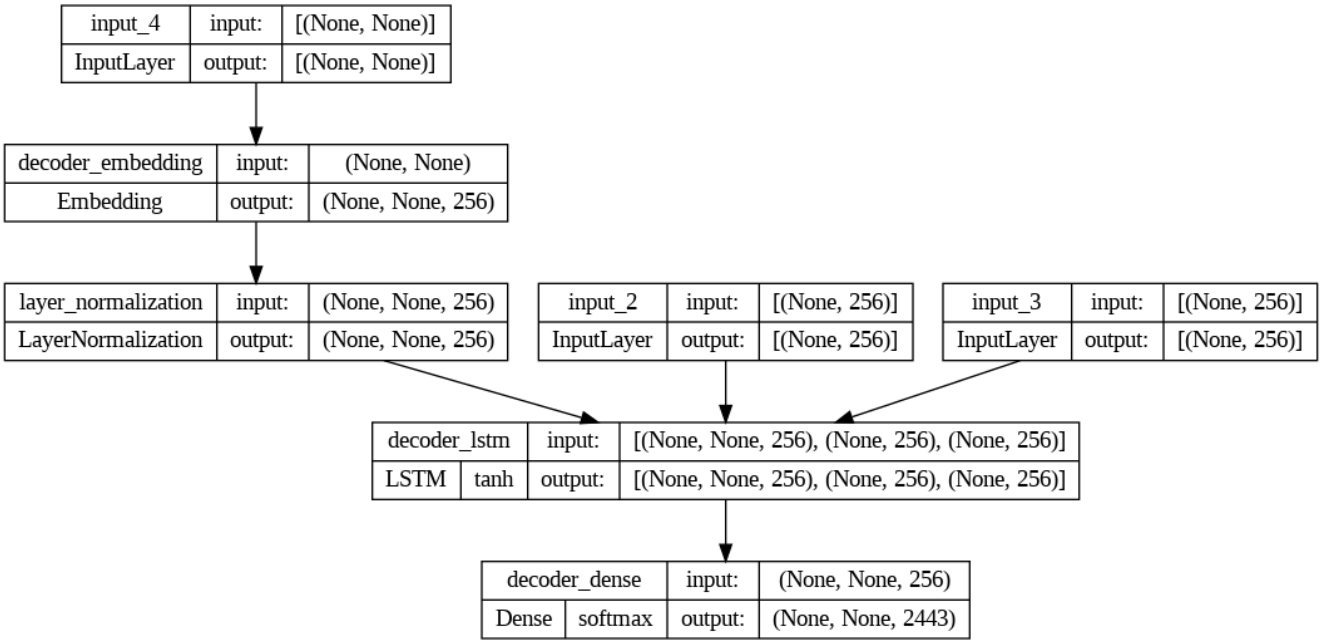
applications.

Simple Visualization for understanding

```
tf.keras.utils.plot_model(chatbot.encoder,to_file='encoder.png',show_shapes=True,show_layer_activations=True)
```




```
tf.keras.utils.plot_model(chatbot.decoder,to_file='decoder.png',show_shapes=True,show_layer_activations=True)
```



Simple Testing

```
>>chatbot("Hello how are you?")
Hi, i've been great. what about you?
>>chatbot("I am good")
Nice to know
^C
```

With That the Chatbot is ready!

Let us Combine this to the Web UI

Integrating the Chatbot with Web UI

Setting-Up Flask

Checking the Flask Install

```
python -m flask
```

```
Usage: python -m flask [OPTIONS] COMMAND [ARGS]...

  A general utility script for Flask applications.

  An application to load must be given with the '--app' option,
'FLASK_APP'
```

```
  environment variable, or with a 'wsgi.py' or 'app.py' file in the
current
  directory.

Options:
  -e, --env-file FILE   Load environment variables from this file.
python-
                        dotenv must be installed.
  -A, --app IMPORT      The Flask application or factory function to
load, in
                        the form 'module:name'. Module can be a dotted
import
                        or file path. Name is not required if it is
'app',
                        'application', 'create_app', or 'make_app', and
can be
                        'name(args)' to pass arguments.
  --debug / --no-debug  Set debug mode.
  --version             Show the Flask version.
  --help                Show this message and exit.

Commands:
  routes  Show the routes for the app.
  run     Run a development server.
  shell   Run a shell in the app context.
```

As we can see the flask is properly installed.

We just need to add the below line at the end to get the Server runnig

```
# Starting an Web Interface
app = Flask(__name__)

@app.route("/")
def home():
    return render_template("index.html")

@app.route('/get_bot_response', methods=['POST'])
def get_bot_response():
    user_message = request.json['user_message']
    bot_response = generate_bot_response(user_message)
```

```
    return jsonify({'bot_response': bot_response})

def generate_bot_response(user_message):
    return chatbot(user_message)

app.run(debug=True)
```

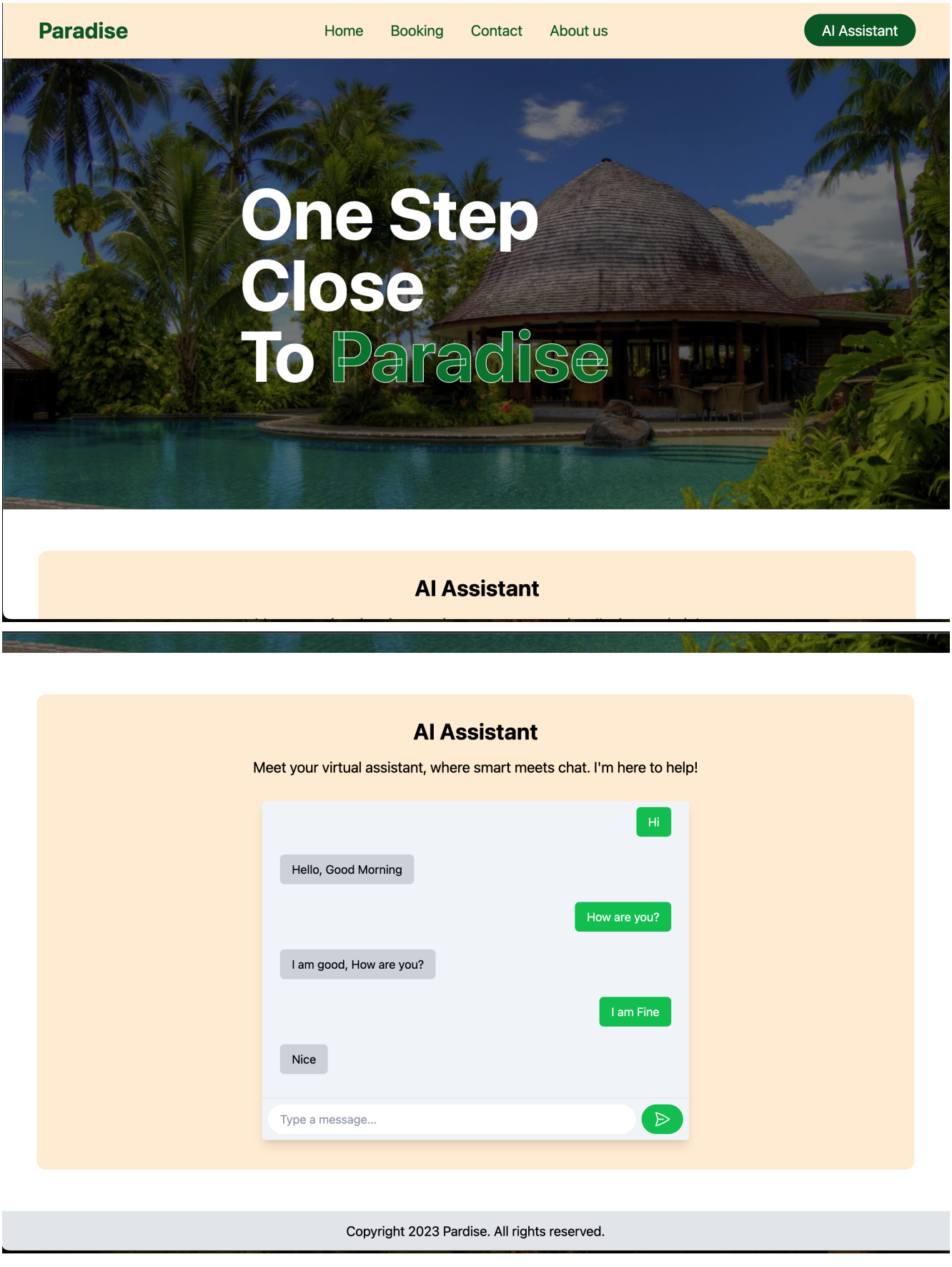
Before we run the flask we need to make sure the file tree is as follows

```
Chatbot
├── bin
├── ckpt
│   ├── assets
│   ├── fingerprint.pb
│   ├── keras_metadata.pb
│   ├── saved_model.pb
│   └── variables
├── dialogs.txt
├── lib
│   ├── train
│   └── validation
├── main.py
├── models
│   ├── assets
│   ├── fingerprint.pb
│   ├── keras_metadata.pb
│   ├── saved_model.pb
│   └── variables
├── node_modules
│   :
├── package-lock.json
├── package.json
├── pyvenv.cfg
├── share
│   └── man
├── static
│   ├── hotel.jpg
│   ├── index.js
│   ├── style.css
│   └── tailwind.css
```

```
├── JS tailwind.config.js
└── templates
    └── index.html
```

Now we can Run the Server

```
┌──  ~/Chatbot
└─ ✓  Chatbot  23:21:10
└─ python3 -i main.py
 * Serving Flask app 'main2'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production
deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 143-143-716
```



And Thus We have Successfully built an Working Hotel Assistant Chatbot

Conclusion