IBM Naanmudhalvan

CHATBOT CREATION USING PYTHON

TEAM MEMBER

211121205023:Raghul S

Phase 5 Project Documentation & Submission

Project: Create Chatbot using Python

Introduction:

A **chatbot** *is "a computer program designed to simulate conversation with human users, especially over the internet."*

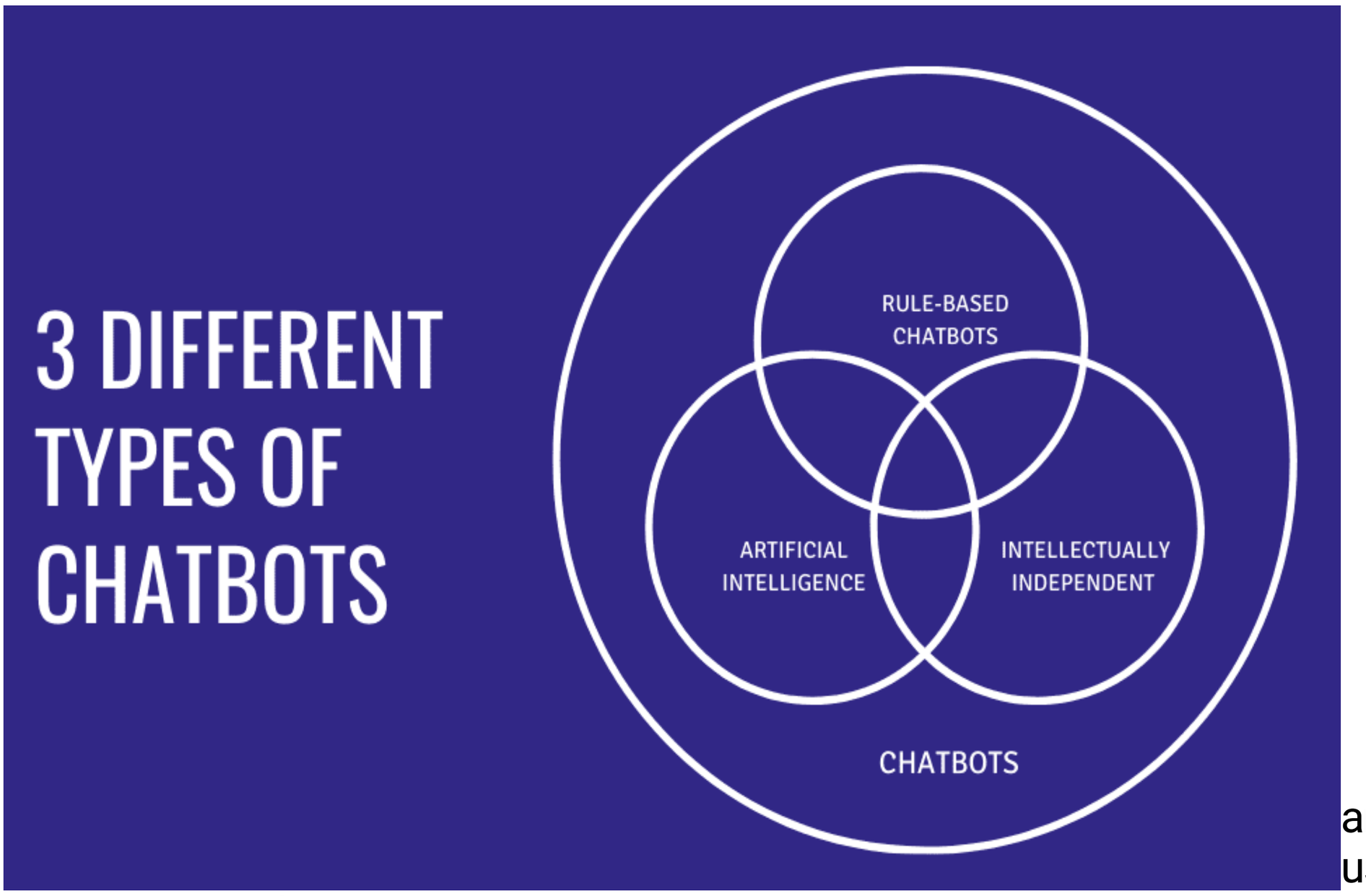
For example, let's say you wish to buy some shoes from a local retailer's website. However, you had some reservations when looking. If the business had an **online chatbot,** you could have gotten answers to all of your questions right away instead of sending long texts.

According to Mordor Intelligence, you can expect the worldwide chatbot industry to increase at a compound annual growth rate of 35 percent from 2021 to 2028, reaching $102 billion. Chatbots have several advantages, including that, unlike apps, they do not need to be downloaded. You don't need to update them, and they don't take up any memory on the

phone. Another advantage is that we may have many bots in the same conversation. This way, we wouldn't have to go from one program to the next, depending on what we need.

Types of Chatbot:

There are three **types of chatbots** in today's digital realm in broad terms. These are:



We are going to use rule-based chatbot for handling customer services, decision-tree bots are another name for rule-based chatbots. As the name states, they follow a set of given rules. These guidelines serve as the foundation for the sorts of problems that the chatbot is familiar with and can solve.

Rule-based chatbots plot out talks like a flowchart. They already have a set of questions to ask the customers they are trained to answer. The customer can choose between those questions and keep moving ahead.

There are either simple or complex rules that are used in rule-based chatbots. However, they can't answer any inquiries that aren't in line with the established guidelines. Interactions do not teach these chatbots anything. Furthermore, they can only execute in specific circumstances for which they have been prepared.

While rule-based bots have a less flexible conversational flow, these safety nets are also beneficial. Chatbots that use machine learning are less predictable, so you can be more certain about the experience you'll get from them.

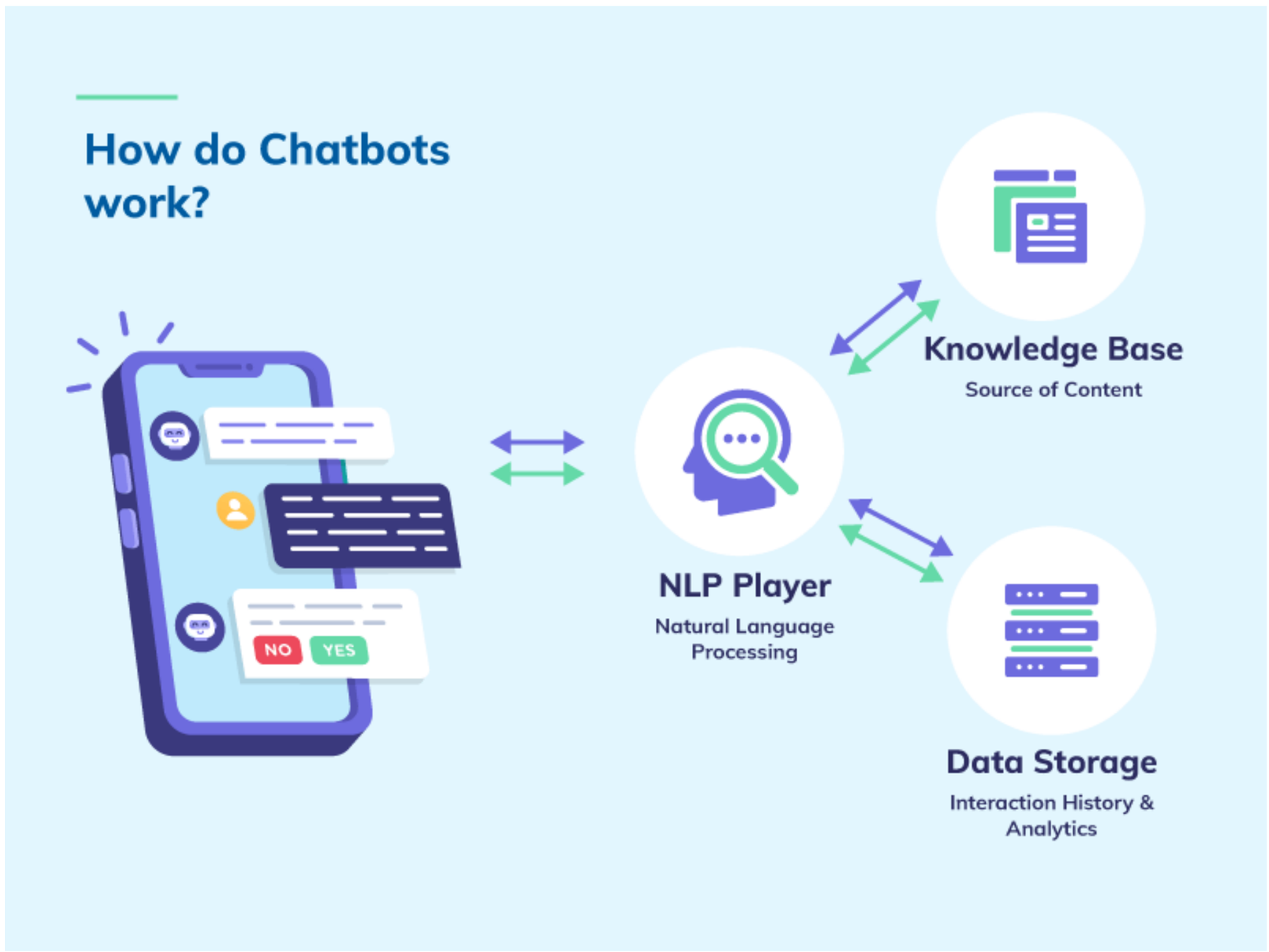
Advantages of Role-based chatbot:

- Easier to train in general (less expensive).
- It's simple to integrate with legacy systems.
- Streamline the transition from a computer to a human agent
- Extremely dependable and safe.
- Interactive components and media can be included.
- Aren't limited to text-based exchanges.

How do chatbot works:

Converting text or speech into structured data is a process for chatbots, especially when programmed to process natural language. Regardless, it's a process that has to be done to give users a proper response to their questions and concerns.

Here is how natural language processing may work with chatbots:



- Some chatbots use tokenization to divide certain words into pieces – or, in this case, “tokens” – that can be pretty useful or significant for the application.
- Named entity recognition looks for categories of words, like the name of a product, a user’s name, or an address, and the chatbot will know what those entities are when the user enters that information in the chatbox.
- Normalization processes text so that it finds common spelling or typographical errors that could happen whenever a user creates a typo or doesn’t know how it spells a particular word.
- Speech tagging allows the chatbot to identify parts of speech such as nouns, verbs, etc. This is so that the chatbot can understand complex sentence structures and how they impact meaning.
- Dependency parsing helps chatbots look for subjects and objects in a given text, leading them to dependent phrases.
- Sentiment analysis lets chatbots watch and learn if a user has a good experience or if they still need help, as compared to a human agent.

Data Source:

- Depending on the particular use case and the needs of the chatbot, many data sources can be used to train AI chatbots. Here is the link to the dataset that will be taken into account for building a Python chatbot that offers first-rate customer support and responds to user inquiries on a website or application.
- Dataset
  Link: https://www.kaggle.com/datasets/grafstor/simple-dialogs-for-chatbot

5

```
hi, how are you doing?  i'm fine. how about yourself?
i'm fine. how about yourself?   i'm pretty good. thanks for asking.
i'm pretty good. thanks for asking.     no problem. so how have you been?
no problem. so how have you been?       i've been great. what about you?
i've been great. what about you?        i've been good. i'm in school right now.
i've been good. i'm in school right now.        what school do you go to?
what school do you go to?       i go to pcc.
i go to pcc.    do you like it there?
do you like it there?   it's okay. it's a really big campus.
it's okay. it's a really big campus.    good luck with school.
good luck with school.  thank you very much.
how's it going? i'm doing well. how about you?
i'm doing well. how about you?  never better, thanks.
never better, thanks.   so how have you been lately?
so how have you been lately?    i've actually been pretty good. you?
i've actually been pretty good. you?    i'm actually in school right now.
i'm actually in school right now.       which school do you attend?
which school do you attend?     i'm attending pcc right now.
i'm attending pcc right now.    are you enjoying it there?
are you enjoying it there?      it's not bad. there are a lot of people there.
it's not bad. there are a lot of people there.  good luck with that.
good luck with that.    thanks.
```

Problem Statement:

Problem:
The challenge is to create a chatbot in Python that provides exceptional customer service, answering user queries on a website or application. The objective is to deliver high-quality support to users, ensuring a positive user experience and customer satisfaction.

Design Thinking Process:

Empathize:

Understand your target audience: Begin by identifying your target users, their preferences, and pain points when it comes to customer service.
Gather feedback: Collect insights from your current customer support team or existing support channels to understand the most common user queries and issues.

Define:

Set clear objectives: Define the specific goals and objectives for your chatbot. What problems will it solve, and what benefits will it provide to users and your organization?
Identify key features: List the essential features your chatbot should have, such as answering frequently asked questions, providing product information, or assisting with troubleshooting.

Ideate:

Brainstorm solutions: Engage your team in brainstorming sessions to generate ideas for how the chatbot can address user needs and deliver exceptional support.
Explore technology options: Investigate the Python libraries and frameworks available for building chatbots, such as Chatterbot, NLTK, or Rasa.

Prototype:

Create a chatbot prototype: Develop a basic prototype of your chatbot using Python. This prototype should focus on a few core features and interactions.
Test internally: Conduct internal testing to refine the chatbot's functionality and identify any issues that need to be addressed.

Test:

User testing: Invite a select group of users to test the chatbot. Collect feedback on the user experience, identify areas for improvement, and assess whether it meets their needs.
Iterative development: Continuously update and refine the chatbot based on user feedback, fixing bugs and enhancing its capabilities.

Implement:

Develop the full chatbot: Based on the feedback from testing, build the complete chatbot using Python. Ensure that it can handle a wide range of user queries and is integrated into your website or application.
Ensure scalability: Plan for the chatbot to handle increasing loads as the user base grows.

Launch:

Deploy the chatbot: Integrate the chatbot into your website or application, making it accessible to users. Communicate the chatbot's availability to your customers.
Monitor performance: Continuously monitor the chatbot's performance, including response times and user satisfaction.

Learn:

Analyze data: Gather data on chatbot interactions, user satisfaction, and the effectiveness of the support it provides.
Iterate and improve: Use the insights gained to make regular improvements to the chatbot's functionality and user experience.

Support and Maintain:

Provide ongoing support: Maintain and update the chatbot to address changing user needs, fix issues, and adapt to new technologies and trends.
Train your customer support team: Ensure that your support team can work effectively with the chatbot, escalate issues when necessary, and provide consistent support.

By following this design thinking process, we can create a customer service chatbot in Python that not only meets user needs but also delivers exceptional customer support, resulting in high user satisfaction.

Phases of development:

Development Part 1- loading and pre-processing the dataset:

Loading the Dataset:

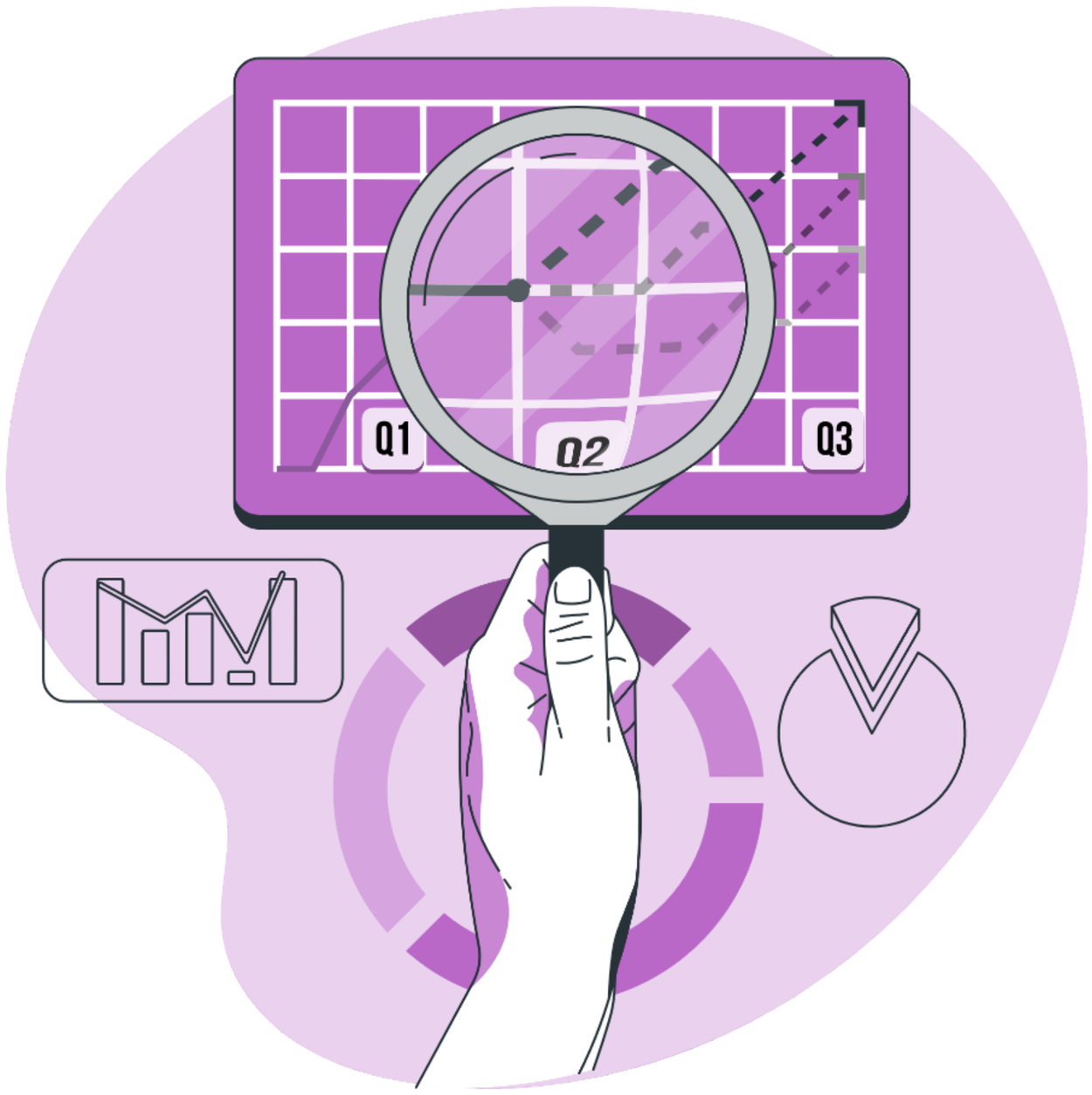
Choose or Create a Dataset:

The first step is to obtain a dataset that suits your chatbot's customer service domain. This dataset should contain pairs of user queries and corresponding bot responses. You can either collect this data manually or explore online datasets or APIs.

- Dataset
  Link: https://www.kaggle.com/datasets/grafstor/simple-dialogs-for-chatbot

Pre-processing the Dataset:

Preprocessing the data is an essential step in data analysis and machine learning. It involves preparing and cleaning the data to make it suitable for analysis or training a model. Here are some common data preprocessing tasks:



Data Cleaning:

Handling missing values by imputing them or removing rows/columns with missing data.
Removing duplicates to ensure data integrity.
Correcting data errors and inconsistencies.

Data Transformation:

Scaling features to have a consistent range, such as using Min-Max scaling or Standardization (z-score normalization).

Encoding categorical variables into numerical values, for example, using one-hot encoding or label encoding.
Handling outliers, either by removing them or transforming them.

Feature Selection:

Identifying and selecting the most relevant features that have a significant impact on the target variable.
Removing irrelevant or redundant features to simplify the model and reduce dimensionality.

Data Splitting:

Splitting the data into training and testing sets to evaluate the model's performance.
Using cross-validation for more robust model assessment.

Handling Imbalanced Data:

Addressing class imbalance issues in classification problems by oversampling, under sampling, or using synthetic data generation techniques.

Text Data Preprocessing:

Tokenization: Breaking text into words or phrases (tokens).
Removing stop words, punctuation, and special characters.
Stemming or lemmatization to reduce words to their base form.

Time Series Data Preprocessing:

Resampling data to different time frequencies (e.g., daily to monthly).
Handling missing time steps.
Feature engineering to create time-based features.

Normalization:

Scaling numerical features to have a mean of 0 and a standard deviation of 1.
Normalizing data for neural networks, such as image pixel values.

Handling Noisy Data:

Detecting and dealing with outliers, anomalies, or data errors.
Smoothing noisy time series data using moving averages or other techniques.

Program:

Import Necessary Libraries:

```
#importing libraries

import numpy as np
import pandas as pd
import os
import warnings
import re
import plotly.express as px
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
from tensorflow.keras.layers import Input, LSTM, Dense, Embedding
from tensorflow.keras.models import Model
from tensorflow.keras.utils import to_categorical
import tensorflow as tf
```

numpy (import numpy as np):

NumPy is a fundamental package for scientific computing in Python.
It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently.
You've imported it using the alias "np" to make it easier to reference in your code.

pandas (import pandas as pd):

11

Pandas is a popular data manipulation and analysis library in Python.
It provides data structures such as DataFrames for handling structured data like spreadsheets or SQL tables.
You've imported it using the alias "pd" for easier usage.

os (import os):

The os module provides a way of interacting with the operating system, allowing you to perform various system-related tasks like file and directory operations.

warnings (import warnings):

The warnings module allows you to control warning messages in your code. You can filter or suppress specific warnings.

re (import re):

The re module is Python's regular expression library, which is used for pattern matching and manipulation of strings.

plotly.express (import plotly.express as px):

Plotly Express is a high-level data visualization library that simplifies the process of creating interactive plots and charts.

tensorflow.keras.preprocessing.text:

This module from TensorFlow's Keras API provides tools for text preprocessing, including Tokenizer for text tokenization and sequence padding.

tensorflow.keras.preprocessing.sequence:

Part of TensorFlow's Keras API, this module deals with sequences and padding for sequences, often used in natural language processing tasks.

sklearn.model_selection:

The model_selection module from scikit-learn (sklearn) provides tools for splitting data into training and testing sets and for other model selection tasks.

tensorflow.keras.layers, tensorflow.keras.models, and tensorflow.keras.utils:

These are modules from TensorFlow's Keras API for building and training neural networks. They include layers for defining network architectures, models for creating neural network models, and utilities for common tasks like one-hot encoding.

tensorflow (import tensorflow as tf):

TensorFlow is a popular deep learning library for building and training machine learning models, including neural networks.

```
# Set up initial configurations
warnings.filterwarnings('ignore')
```

Step 1: Load the dataset

```
df = pd.read_csv('/kaggle/input/simple-dialogs-for-chatbot/dialogs.txt', sep='\t', names=['Question', 'Answer'])
df
```

Output:

	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.
...	...	...
3720	that's a good question. maybe it's not old age.	are you right-handed?
3721	are you right-handed?	yes. all my life.
3722	yes. all my life.	you're wearing out your right hand. stop using...
3723	you're wearing out your right hand. stop using...	but i do all my writing with my right hand.
3724	but i do all my writing with my right hand.	start typing instead. that way your left hand ...

3725 rows × 2 columns

Step 2: Check for null values and whitespace in the dataset

```
# Data preprocessing and quality checks

null_question = df['Question'].isnull().sum()
null_answer = df['Answer'].isnull().sum()
if null_question > 0:
```

```
    print("There are", null_question, "null values in the 'Question' column.")
else:
    print("There are no null values in the 'Question' column.")

if null_answer > 0:
    print("There are", null_answer, "null values in the 'Answer' column.")
else:
    print("There are no null values in the 'Answer' column.")
```

Output:

```
There are no null values in the 'Question' column.
There are no null values in the 'Answer' column.
```

```
# Check for whitespace values
whitespace_question = df['Question'].apply(lambda x: x.isspace()).sum()
whitespace_answer = df['Answer'].apply(lambda x: x.isspace()).sum()

if whitespace_question > 0:
    print("There are", whitespace_question, "whitespace values in the 'Question'
column.")
else:
    print("There are no whitespace values in the 'Question' column.")

if whitespace_answer > 0:
    print("There are", whitespace_answer, "whitespace values in the 'Answer' column.
")
else:
    print("There are no whitespace values in the 'Answer' column.")
```

Output:

```
There are no whitespace values in the 'Question' column.
There are no whitespace values in the 'Answer' column.
```

Step 4: Define a text cleaning function

```
# Clean and preprocess text data

def clean_text(text):
    text = text.lower()
    text = re.sub(r'\d+', ' ', text)
```

```
    text = re.sub(r'([^\w\s])', r' \1 ', text)
    text = re.sub(r'\s+', ' ', text)
    text = text.strip()
    return text
df['Encoder Inputs'] = df['Question'].apply(clean_text)
df['Decoder Inputs'] = "<sos> " + df['Answer'].apply(clean_text) + ' <eos>'
df["Decoder Targets"] = df['Answer'].apply(clean_text) + ' <eos>'
df.head()
```

Output:

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

Step 5: Analyze the length distribution of questions and answers

```
df['Question Length'] = df['Encoder Inputs'].apply(lambda x: len(x))
df['Answer Length'] = df['Decoder Inputs'].apply(lambda x: len(x))

df.head()
```

15

	Question	Answer	Encoder Inputs	Decoder Inputs	Decoder Targets	Question Length	Answer Length
0	hi, how are you doing?	i'm fine. how about yourself?	hi , how are you doing ?	<sos> i ' m fine . how about yourself ? <eos>	i ' m fine . how about yourself ? <eos>	24	45
1	i'm fine. how about yourself?	i'm pretty good. thanks for asking.	i ' m fine . how about yourself ?	<sos> i ' m pretty good . thanks for asking	i ' m pretty good . thanks for asking . <eos>	33	51
2	i'm pretty good. thanks for asking.	no problem. so how have you been?	i ' m pretty good . thanks for asking .	<sos> no problem . so how have you been ? <eos>	no problem . so how have you been ? <eos>	39	47
3	no problem. so how have you been?	i've been great. what about you?	no problem . so how have you been ?	<sos> i ' ve been great . what about you ? <eos>	i ' ve been great . what about you ? <eos>	35	48
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 ?	<sos> i ' ve been good . i ' m in school right...	i ' ve been good . i ' m in school right now	36	58

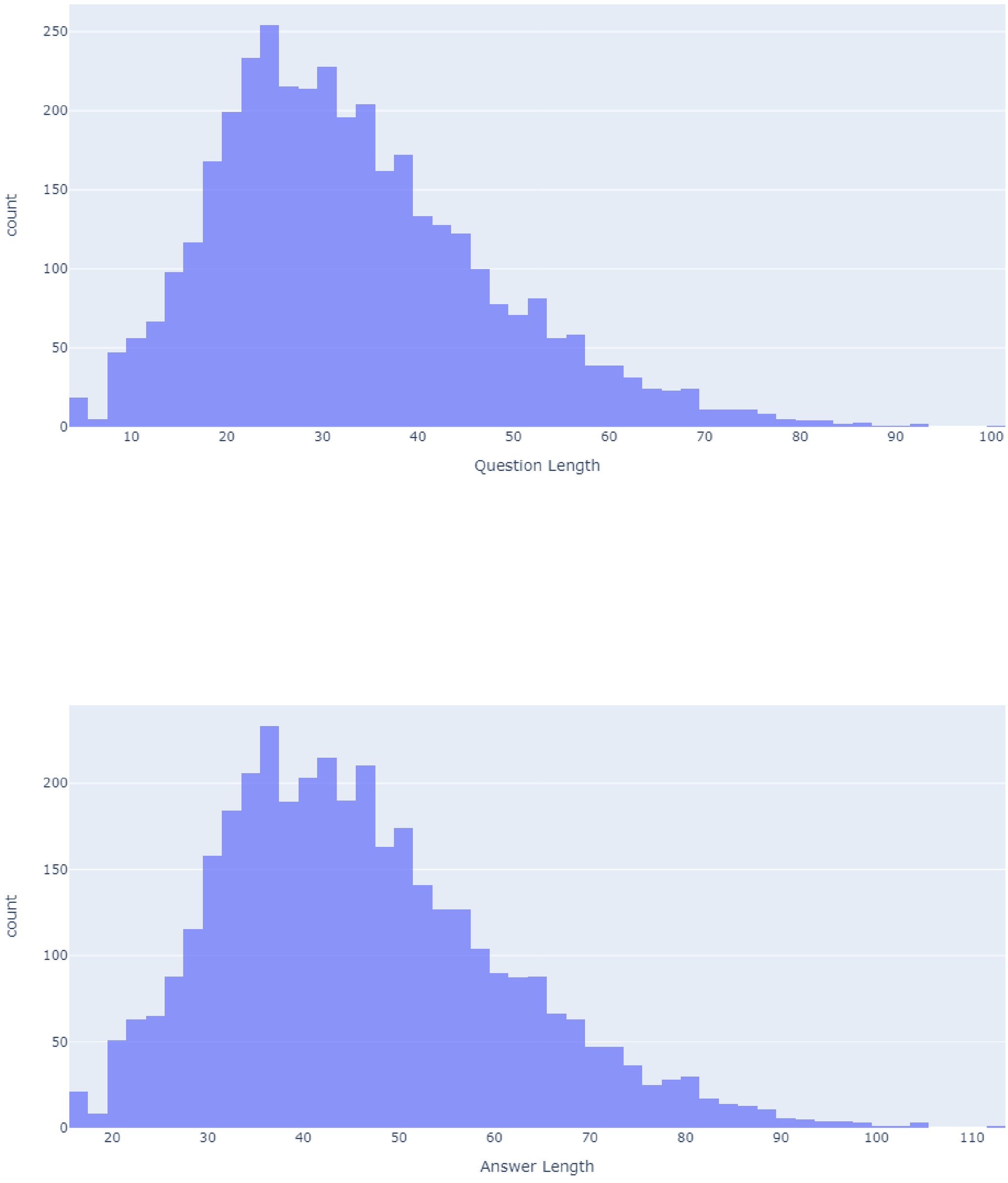
Visualize:

```
import plotly.express as px
fig1 = px.histogram(df, x='Question Length', nbins=50, opacity=0.7)
fig2 = px.histogram(df, x='Answer Length', nbins=50, opacity=0.7)

print("Maximum Question Length:", df['Question Length'].max())
print("Maximum Answer Length:", df['Answer Length'].max())

fig1.show()
fig2.show()
```

Output:

```
Maximum Question Length: 101
Maximum Answer Length: 113
```

16



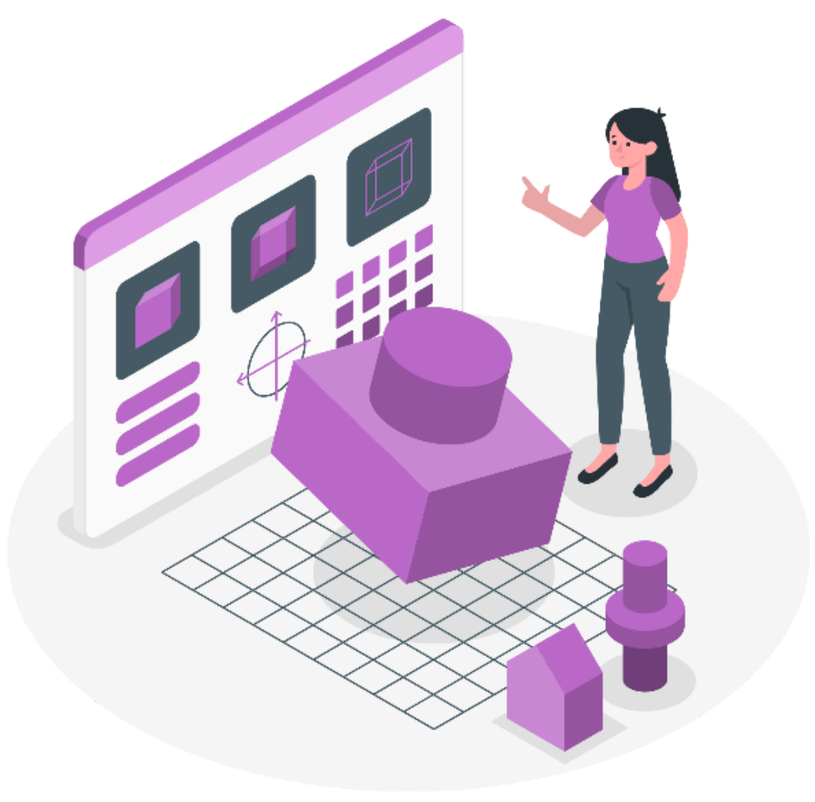
Step 6: Tokenization and Padding

```
num_words = 10000
max_seq_length = 10

tokenizer = Tokenizer(num_words=num_words, oov_token='<unk>')
tokenizer.fit_on_texts(df['Encoder Inputs'].tolist() + df['Decoder Inputs'].tolist())

encoder_inputs = tokenizer.texts_to_sequences(df['Encoder Inputs'].tolist())
decoder_inputs = tokenizer.texts_to_sequences(df['Decoder Inputs'].tolist())
decoder_targets = tokenizer.texts_to_sequences(df['Decoder Targets'].tolist())

encoder_inputs = pad_sequences(encoder_inputs, maxlen=max_seq_length,
padding='post', truncating='post')
decoder_inputs = pad_sequences(decoder_inputs, maxlen=max_seq_length,
padding='post', truncating='post')
decoder_targets = pad_sequences(decoder_targets, maxlen=max_seq_length,
padding='post', truncating='post')

df['Decoder Targets'][1:3]
```

Output:

```
1    i ' m pretty good . thanks for asking . <eos>
2       no problem . so how have you been ? <eos>
Name: Decoder Targets, dtype: object
```

Training Model:



Step 7: Split the data into train and test sets

encoder_inputs_train, encoder_inputs_test, decoder_inputs_train, decoder_inputs_test, decoder_targets_train, decoder_targets_test = train_test_split(encoder_inputs, decoder_inputs, decoder_targets, test_size=0.2, random_state=42)

Step 8: Define the Seq2Seq model architecture
Sequence-to-Sequence Model: This architecture is commonly used for a variety of natural language processing applications, such as chatbot generation and language translation, which is why we choose it. There are two primary parts to this model: an encoder and a decoder.

LSTM (Long Short-Term Memory): We choose to use LSTM layers in the encoder and decoder of the sequence-to-sequence model. LSTM cells are useful for comprehending discussions and producing well-reasoned answers because of their reputation for capturing long-range relationships in sequences.

Neural Network Structure:

The neural network structure of our chatbot model can be summarized as follows:

Encoder:

Embedding Layer: The input text is first passed through an embedding layer, which converts words into dense vectors. These vectors are trainable and help the model understand the meaning of words in the context of the conversation.
LSTM Layers: After embedding, the data is fed into one or more LSTM layers in the encoder. These layers process the input sequence and encode it into a fixed-size context vector, summarizing the input information.

Decoder:

LSTM Layers: The decoder also consists of one or more LSTM layers. These layers take the context vector from the encoder and generate the response sequence one word at a time.
Dense Layer: A dense layer with a softmax activation function is used to predict the next word in the response sequence at each time step.

The model is trained to minimize the loss between the predicted response and the actual response in the training data. This way, it learns to generate coherent and contextually relevant responses in conversations.

```
num_encoder_tokens = len(tokenizer.word_index) + 1
num_decoder_tokens = len(tokenizer.word_index) + 1
latent_dim = 32
embedding_dim = 50

encoder_inputs = Input(shape=(max_seq_length,))
encoder_embedding = Embedding(num_encoder_tokens, embedding_dim,
mask_zero=True)
encoder_inputs_embedded = encoder_embedding(encoder_inputs)

encoder_lstm1 = LSTM(latent_dim, return_sequences=True, return_state=True,
dropout=0.4, recurrent_dropout=0.4)
encoder_output1, state_h1, state_c1 = encoder_lstm1(encoder_inputs_embedded)

encoder_lstm2 = LSTM(latent_dim, return_sequences=True, return_state=True,
dropout=0.4, recurrent_dropout=0.4)
encoder_output2, state_h2, state_c2 = encoder_lstm2(encoder_output1)

encoder_lstm3 = LSTM(latent_dim, return_state=True, return_sequences=True,
dropout=0.4, recurrent_dropout=0.4)
encoder_outputs, state_h, state_c = encoder_lstm3(encoder_output2)

encoder_states = [state_h, state_c]

decoder_inputs = Input(shape=(max_seq_length,))
decoder_embedding = Embedding(num_decoder_tokens, embedding_dim,
mask_zero=True)
decoder_inputs_embedded = decoder_embedding(decoder_inputs)

decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_inputs_embedded,
initial_state=encoder_states)

decoder_dense = Dense(num_decoder_tokens, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)
```

20

model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

Explanation:

Tokenization and Vocabulary Size:
num_encoder_tokens and num_decoder_tokens are calculated based on the size of the vocabulary, which is the number of unique tokens in the source and target languages. The +1 is added to account for a special token or padding.

Model Parameters:
latent_dim is the dimensionality of the LSTM's hidden state.
embedding_dim is the dimensionality of the word embeddings.

Encoder:
encoder_inputs is the input layer for the source language sequences, which have a shape of (max_seq_length,).
encoder_embedding is the embedding layer for the encoder inputs, which transforms words into dense vectors.
encoder_lstm1, encoder_lstm2, and encoder_lstm3 are three stacked LSTM layers in the encoder. They are designed to capture sequential information from the input sequences. They return sequences (return_sequences=True) and the final hidden states (return_state=True).
encoder_states store the final hidden states of the encoder. These states will be used as the initial states for the decoder.

Decoder:
decoder_inputs is the input layer for the target language sequences.
decoder_embedding is the embedding layer for the decoder inputs.
decoder_lstm is a single LSTM layer in the decoder. It takes the embedded decoder inputs and the initial states from the encoder and returns sequences and the final hidden state.
decoder_dense is a dense layer with a softmax activation function. It converts the decoder LSTM's outputs into a probability distribution over the vocabulary, which represents the likelihood of the next word in the translation.

Model:
model is created using the Keras Functional API. It takes both the encoder and decoder inputs and produces the decoder outputs.

In a typical Seq2Seq model for machine translation, you would train the model with parallel source and target sequences. The encoder processes the source sequence, and the decoder generates the target sequence. During training, the model is trained to minimize the difference between the predicted sequence and the actual target sequence. The model is then used for inference by providing a source sequence and generating a target sequence.

Step 9: Compile and train the model

```
early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3)
batch_size = 32
epochs = 30

decoder_targets_train = to_categorical(decoder_targets_train,
num_decoder_tokens)
decoder_targets_test = to_categorical(decoder_targets_test, num_decoder_tokens)

model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'], sample_weight_mode='temporal')

model.fit([encoder_inputs_train, decoder_inputs_train], decoder_targets_train,
        validation_data=([encoder_inputs_test, decoder_inputs_test],
decoder_targets_test),
        batch_size=batch_size, epochs=epochs, callbacks=[early_stopping])
```

Output:

```
Epoch 1/30
94/94 [==============================] - 47s 312ms/step - loss: 6.7108 -
accuracy: 0.0869 - val_loss: 5.3961 - val_accuracy: 0.0928
Epoch 2/30
94/94 [==============================] - 20s 215ms/step - loss: 5.3310 -
accuracy: 0.0943 - val_loss: 5.3034 - val_accuracy: 0.0930
Epoch 3/30
94/94 [==============================] - 20s 212ms/step - loss: 5.2223 -
accuracy: 0.1047 - val_loss: 5.1698 - val_accuracy: 0.1280
Epoch 4/30
94/94 [==============================] - 18s 187ms/step - loss: 5.0835 -
accuracy: 0.1332 - val_loss: 5.0813 - val_accuracy: 0.1463
```

22

```
Epoch 5/30
94/94 [==============================] - 18s 197ms/step - loss: 4.9987 -
accuracy: 0.1468 - val_loss: 5.0264 - val_accuracy: 0.1688
Epoch 6/30
94/94 [==============================] - 18s 193ms/step - loss: 4.9282 -
accuracy: 0.1704 - val_loss: 4.9702 - val_accuracy: 0.1929
Epoch 7/30
94/94 [==============================] - 19s 200ms/step - loss: 4.8579 -
accuracy: 0.1966 - val_loss: 4.9100 - val_accuracy: 0.2087
Epoch 8/30
94/94 [==============================] - 17s 182ms/step - loss: 4.7839 -
accuracy: 0.2178 - val_loss: 4.8454 - val_accuracy: 0.2234
Epoch 9/30
94/94 [==============================] - 18s 194ms/step - loss: 4.7069 -
accuracy: 0.2323 - val_loss: 4.7812 - val_accuracy: 0.2352
Epoch 10/30
94/94 [==============================] - 18s 193ms/step - loss: 4.6269 -
accuracy: 0.2407 - val_loss: 4.7146 - val_accuracy: 0.2376
Epoch 11/30
94/94 [==============================] - 17s 183ms/step - loss: 4.5527 -
accuracy: 0.2438 - val_loss: 4.6623 - val_accuracy: 0.2394
Epoch 12/30
94/94 [==============================] - 18s 189ms/step - loss: 4.4903 -
accuracy: 0.2458 - val_loss: 4.6196 - val_accuracy: 0.2398
Epoch 13/30
94/94 [==============================] - 18s 189ms/step - loss: 4.4336 -
accuracy: 0.2544 - val_loss: 4.5823 - val_accuracy: 0.2535
Epoch 14/30
94/94 [==============================] - 19s 200ms/step - loss: 4.3840 -
accuracy: 0.2640 - val_loss: 4.5514 - val_accuracy: 0.2584
Epoch 15/30
94/94 [==============================] - 17s 183ms/step - loss: 4.3383 -
accuracy: 0.2670 - val_loss: 4.5223 - val_accuracy: 0.2597
Epoch 16/30
94/94 [==============================] - 18s 193ms/step - loss: 4.2965 -
accuracy: 0.2682 - val_loss: 4.4970 - val_accuracy: 0.2631
Epoch 17/30
94/94 [==============================] - 18s 196ms/step - loss: 4.2580 -
accuracy: 0.2707 - val_loss: 4.4717 - val_accuracy: 0.2650
```

23

```
Epoch 18/30
94/94 [==============================] - 18s 187ms/step - loss: 4.2189 -
accuracy: 0.2741 - val_loss: 4.4494 - val_accuracy: 0.2665
Epoch 19/30
94/94 [==============================] - 18s 194ms/step - loss: 4.1817 -
accuracy: 0.2757 - val_loss: 4.4274 - val_accuracy: 0.2689
Epoch 20/30
94/94 [==============================] - 18s 187ms/step - loss: 4.1472 -
accuracy: 0.2781 - val_loss: 4.4103 - val_accuracy: 0.2681
Epoch 21/30
94/94 [==============================] - 18s 191ms/step - loss: 4.1131 -
accuracy: 0.2809 - val_loss: 4.3946 - val_accuracy: 0.2725
Epoch 22/30
94/94 [==============================] - 17s 184ms/step - loss: 4.0805 -
accuracy: 0.2855 - val_loss: 4.3810 - val_accuracy: 0.2747
Epoch 23/30
94/94 [==============================] - 18s 193ms/step - loss: 4.0487 -
accuracy: 0.2907 - val_loss: 4.3662 - val_accuracy: 0.2791
Epoch 24/30
94/94 [==============================] - 17s 183ms/step - loss: 4.0189 -
accuracy: 0.2961 - val_loss: 4.3577 - val_accuracy: 0.2832
Epoch 25/30
94/94 [==============================] - 18s 186ms/step - loss: 3.9895 -
accuracy: 0.3002 - val_loss: 4.3448 - val_accuracy: 0.2856
Epoch 26/30
94/94 [==============================] - 18s 187ms/step - loss: 3.9640 -
accuracy: 0.3042 - val_loss: 4.3356 - val_accuracy: 0.2875
Epoch 27/30
94/94 [==============================] - 17s 181ms/step - loss: 3.9346 -
accuracy: 0.3067 - val_loss: 4.3241 - val_accuracy: 0.2887
Epoch 28/30
94/94 [==============================] - 18s 190ms/step - loss: 3.9099 -
accuracy: 0.3096 - val_loss: 4.3183 - val_accuracy: 0.2903
Epoch 29/30
94/94 [==============================] - 18s 187ms/step - loss: 3.8859 -
accuracy: 0.3105 - val_loss: 4.3081 - val_accuracy: 0.2914
Epoch 30/30
94/94 [==============================] - 18s 189ms/step - loss: 3.8610 -
accuracy: 0.3127 - val_loss: 4.3015 - val_accuracy: 0.2919
```

```
# Helper function to generate responses
def generate_response(input_seq):
    states_value = encoder_model.predict(input_seq)
    target_seq = np.array([[tokenizer.word_index['sos']]]
    stop_condition = False
    response = []

    while not stop_condition:
        output_tokens, h, c = decoder_model.predict([target_seq] + states_value)
        sampled_token_index = np.argmax(output_tokens[0, -1, :])

        if sampled_token_index == 0:
            sampled_token = '.'
        else:
            sampled_token = tokenizer.index_word[sampled_token_index]

        response.append(sampled_token)

        if sampled_token == 'eos' or len(response) > max_seq_length:
            stop_condition = True

        target_seq = np.array([[sampled_token_index]])
        states_value = [h, c]

    return ' '.join(response)

# Test the chatbot
# Preprocess and tokenize the user's input
user_input = "Hello, how are you?"
input_sequence = tokenizer.texts_to_sequences([user_input])
input_sequence = pad_sequences(input_sequence, maxlen=max_seq_length,
padding='post', truncating='post')

# Generate a response
response = generate_response(input_sequence)

1/1 [==============================] - 1s 839ms/step
1/1 [==============================] - 2s 2s/step
```

25

```
1/1 [==============================] - 0s 35ms/step
1/1 [==============================] - 0s 39ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 43ms/step
1/1 [==============================] - 0s 36ms/step
1/1 [==============================] - 0s 51ms/step
1/1 [==============================] - 0s 37ms/step
1/1 [==============================] - 0s 37ms/step
Input: [[117 110   0   0   0   0   0   0   0   0]]
Response: i ' s a lot of the lot of the lot
```

Development Part 1- Web Development:

Start building the chatbot by integrating it into a web app using Flask

Web App Development with Flask

- Flask is a micro web framework for Python that is widely used for web app development. It's known for its simplicity and flexibility, making it an excellent choice for developing web applications of various sizes and complexities.

Installation:

First, make sure you have Python installed. You can then install Flask using pip:

pip install Flask

Project Structure:

Organize your project by creating a directory structure. A basic structure might look like this:

```
my_flask_app/
    ├── app.py
    ├── templates/
    │   └── index.html
```

```
└── static/
└── style.css
```

app.py

Program;

```
import numpy as np
import pandas as pd
import os
import warnings
import re
from flask import Flask, render_template, request, jsonify
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Model
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.layers import Input, LSTM, Dense, Embedding
import tensorflow as tf
from sklearn.model_selection import train_test_split


app = Flask(__name__)


# Set up initial configurations
warnings.filterwarnings('ignore')


# Step 1: Load the dataset and import necessary libraries
df = pd.read_csv('/kaggle/input/database/dialogs.txt', sep='\t',
```

```
names=['Question', 'Answer'])


# Step 2: Check for null values and whitespace in the dataset
# Data preprocessing and quality checks
null_question = df['Question'].isnull().sum()
null_answer = df['Answer'].isnull().sum()


# Step 3: Visualize the data distribution
whitespace_question = df['Question'].apply(lambda x: x.isspace()).sum()
whitespace_answer = df['Answer'].apply(lambda x: x.isspace()).sum()


# Step 4: Define a text cleaning function
# Clean and preprocess text data
def clean_text(text):
    text = text.lower()
    text = re.sub(r'\d+', ' ', text)
    text = re.sub(r'([^\w\s])', r' \1 ', text)
    text = re.sub(r'\s+', ' ', text)
    text = text.strip()
    return text


df['Encoder Inputs'] = df['Question'].apply(clean_text)
df['Decoder Inputs'] = "<sos> " + df['Answer'].apply(clean_text) + ' <eos>'
df["Decoder Targets"] = df['Answer'].apply(clean_text) + ' <eos>'
```

28

```
# Step 5: Analyze the length distribution of questions and answers
df['Question Length'] = df['Encoder Inputs'].apply(lambda x: len(x))
df['Answer Length'] = df['Decoder Inputs'].apply(lambda x: len(x))


# Step 6: Tokenization and Padding
num_words = 10000
max_seq_length = 10


tokenizer = Tokenizer(num_words=num_words, oov_token='<unk>')
tokenizer.fit_on_texts(df['Encoder Inputs'].tolist() + df['Decoder Inputs'].tolist())


encoder_inputs = tokenizer.texts_to_sequences(df['Encoder Inputs'].tolist())
decoder_inputs = tokenizer.texts_to_sequences(df['Decoder Inputs'].tolist())
decoder_targets = tokenizer.texts_to_sequences(df['Decoder Targets'].tolist())


encoder_inputs = pad_sequences(encoder_inputs, maxlen=max_seq_length,
padding='post', truncating='post')
decoder_inputs = pad_sequences(decoder_inputs, maxlen=max_seq_length,
padding='post', truncating='post')
decoder_targets = pad_sequences(decoder_targets, maxlen=max_seq_length,
padding='post', truncating='post')


# Step 7: Split the data into train and test sets
encoder_inputs_train, encoder_inputs_test, decoder_inputs_train,
decoder_inputs_test, decoder_targets_train, decoder_targets_test =
train_test_split(encoder_inputs, decoder_inputs, decoder_targets, test_size=0.2,
random_state=42)
```

```
# Step 8: Define the Seq2Seq model architecture
num_encoder_tokens = len(tokenizer.word_index) + 1
num_decoder_tokens = len(tokenizer.word_index) + 1
latent_dim = 32
embedding_dim = 50


encoder_inputs = Input(shape=(max_seq_length,))
encoder_embedding = Embedding(num_encoder_tokens, embedding_dim, mask_zero=True)
encoder_inputs_embedded = encoder_embedding(encoder_inputs)


encoder_lstm1 = LSTM(latent_dim, return_sequences=True, return_state=True, dropout=0.4, recurrent_dropout=0.4)
encoder_output1, state_h1, state_c1 = encoder_lstm1(encoder_inputs_embedded)


encoder_lstm2 = LSTM(latent_dim, return_sequences=True, return_state=True, dropout=0.4, recurrent_dropout=0.4)
encoder_output2, state_h2, state_c2 = encoder_lstm2(encoder_output1)


encoder_lstm3 = LSTM(latent_dim, return_state=True, return_sequences=True, dropout=0.4, recurrent_dropout=0.4)
encoder_outputs, state_h, state_c = encoder_lstm3(encoder_output2)


encoder_states = [state_h, state_c]
```

30

```
decoder_inputs = Input(shape=(max_seq_length,))

decoder_embedding = Embedding(num_decoder_tokens, embedding_dim,
mask_zero=True)

decoder_inputs_embedded = decoder_embedding(decoder_inputs)



decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)

decoder_outputs, _, _ = decoder_lstm(decoder_inputs_embedded,
initial_state=encoder_states)
```

```
decoder_dense = Dense(num_decoder_tokens, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)

model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

# Step 9: Compile and train the model
early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3)
batch_size = 32
epochs = 30
```

```
decoder_targets_train = to_categorical(decoder_targets_train,
num_decoder_tokens)

decoder_targets_test = to_categorical(decoder_targets_test,
num_decoder_tokens)



model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'], sample_weight_mode='temporal')
```

```
model.fit([encoder_inputs_train, decoder_inputs_train], decoder_targets_train,
          validation_data=([encoder_inputs_test, decoder_inputs_test],
decoder_targets_test),
          batch_size=batch_size, epochs=epochs, callbacks=[early_stopping])


# Step 10: Create encoder and decoder models
encoder_model = Model(encoder_inputs, encoder_states)


decoder_state_input_h = Input(shape=(latent_dim,))
decoder_state_input_c = Input(shape=(latent_dim,))
decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]


decoder_inputs_single = Input(shape=(1,))
decoder_inputs_single_embedded =
decoder_embedding(decoder_inputs_single)


decoder_outputs, state_h, state_c =
decoder_lstm(decoder_inputs_single_embedded,
initial_state=decoder_states_inputs)
decoder_states = [state_h, state_c]
decoder_outputs = decoder_dense(decoder_outputs)


decoder_model = Model([decoder_inputs_single] + decoder_states_inputs,
[decoder_outputs] + decoder_states)


# Step 6: Tokenization and Padding
num_words = 10000
```

32

```
max_seq_length = 10


tokenizer = Tokenizer(num_words=num_words, oov_token='<unk>')
tokenizer.fit_on_texts(df['Encoder Inputs'].tolist() + df['Decoder Inputs'].tolist())


encoder_inputs = tokenizer.texts_to_sequences(df['Encoder Inputs'].tolist())
decoder_inputs = tokenizer.texts_to_sequences(df['Decoder Inputs'].tolist())
decoder_targets = tokenizer.texts_to_sequences(df['Decoder Targets'].tolist())


encoder_inputs = pad_sequences(encoder_inputs, maxlen=max_seq_length,
padding='post', truncating='post')

decoder_inputs = pad_sequences(decoder_inputs, maxlen=max_seq_length,
padding='post', truncating='post')

decoder_targets = pad_sequences(decoder_targets, maxlen=max_seq_length,
padding='post', truncating='post')


def generate_response(input_seq):

    states_value = encoder_model.predict(input_seq)

    target_seq = np.array([tokenizer.word_index['sos']])

    stop_condition = False

    response = []


    while not stop_condition:

        output_tokens, h, c = decoder_model.predict([target_seq] + states_value)

        sampled_token_index = np.argmax(output_tokens[0, -1, :])
```

```
        if sampled_token_index == 0:
            sampled_token = '.'
        else:
            sampled_token = tokenizer.index_word[sampled_token_index]

        response.append(sampled_token)

        if sampled_token == 'eos' or len(response) > max_seq_length:
            stop_condition = True

        target_seq = np.array([sampled_token_index])
        states_value = [h, c]

    return ' '.join(response)

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

@app.route('/ask', methods=['POST'])
def ask():
    user_input = request.form['user_input']
    # Tokenize and preprocess the user input
    input_seq = tokenizer.texts_to_sequences([user_input])
    input_seq = pad_sequences(input_seq, maxlen=max_seq_length,
```

```
padding='post', truncating='post')

    # Generate a response
    response = generate_response(input_seq)

    return jsonify({'response': response})

if __name__ == "__main__":
    app.run()
```

In this code, we import Flask and create a Flask app. We define two routes: '/' for the main page and '/chat' to handle user interactions.

- Create a folder named templates in your project directory, and inside it, create an HTML file named index.html. Here's a simple example of what it could look like:

index.html

Program;

```
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
    <div class="chatbot">
        <div class="chat-header">
```

```
    <h2>Chatbot</h2>
</div>
<div class="chat-box" id="chat-box">
    <div class="message received">
        <p>Hello! How can I help you today?</p>
    </div>
</div>
<div class="user-input">
    <input type="text" id="user-input" placeholder="Type a message...">
    <button onclick="sendMessage()">Send</button>
</div>
</body>
<script>
    function sendMessage() {
        const userMessage = document.getElementById("user-input").value;
        if (userMessage === "") return;

        const chatBox = document.getElementById("chat-box");
        const userDiv = document.createElement("div");
        userDiv.className = "message sent";
        userDiv.innerHTML = "<p>" + userMessage + "</p>";

        chatBox.appendChild(userDiv);
```

```
// Send the user's message to the server and receive a response.
fetch("/chat", {
    method: "POST",
    body: JSON.stringify({ user_input: userMessage }),
    headers: {
        "Content-Type": "application/json",
    },
})
.then((response) => response.text())
.then((botResponse) => {
    const botMessage = document.createElement("div");
    botMessage.className = "message received";
    botMessage.innerHTML = "<p>" + botResponse + "</p>";
    chatBox.appendChild(botMessage);
})
.catch((error) => console.error(error));

        document.getElementById("user-input").value = "";
    }
</script>
</html>
```

This code provides a basic chatbot interface where users can input messages, and the chatbot responds with a simulated response. Note that there is room for enhancing the chatbot's functionality and appearance, and the actual chatbot logic is not provided here.

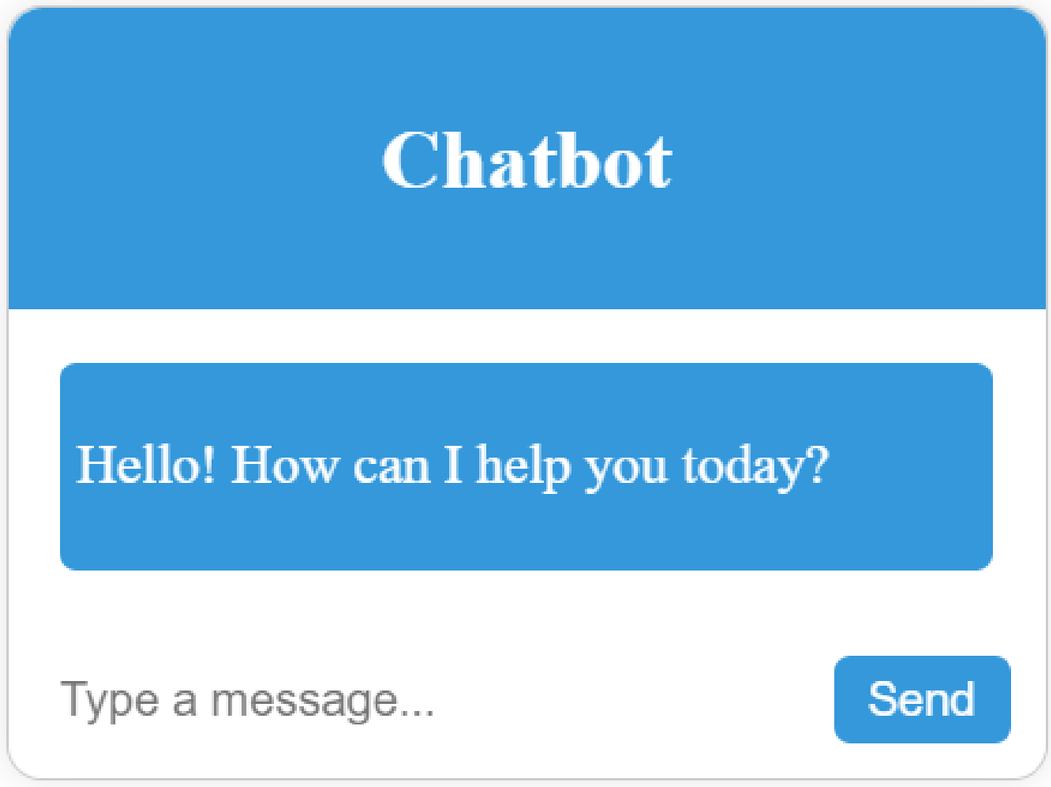
style.css

Program;

```
body {
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    margin: 0;
}

.chatbot {
    width: 300px;
    border: 1px solid #ccc;
    box-shadow: 0px 0px 10px rgba(0, 0, 0, 0.1);
    border-radius: 10px;
}

.chat-header {
    background-color: #3498db;
    color: white;
    text-align: center;
    padding: 10px;
    border-top-left-radius: 10px;
    border-top-right-radius: 10px;
}
```

```
.chat-box {
    padding: 10px;
    max-height: 300px;
    overflow-y: auto;
}

.message {
    padding: 5px;
    margin: 5px;
    border-radius: 5px;
    word-wrap: break-word;
}

.message.sent {
    background-color: #f1f1f1;
    text-align: right;
}

.message.received {
    background-color: #3498db;
    color: white;
}

.user-input {
```

```
        display: flex;
        justify-content: space-between;
        padding: 10px;
    }
    input[type="text"] {
        flex: 1;
        border: none;
        border-radius: 5px;
        padding: 5px;
        margin-right: 10px;
    }
    button {
        background-color: #3498db;
        color: white;
        border: none;
        border-radius: 5px;
        padding: 5px 10px;
        cursor: pointer;
    }
```



Access the chatbot through your web browser at http://localhost:5000.

This Flask web app will allow users to interact with the chatbot through a simple web interface. Users can type their questions, and the chatbot will provide responses based on the data from the data file.

In conclusion, the provided code sets up the foundations for a simple chatbot using a Seq2Seq model and integrates it into a Flask web application for user interaction. However, it's important to note that this code represents a basic starting point for a chatbot and may require additional fine-tuning, data, and more advanced natural language processing techniques to build a fully functional and context-aware chatbot. Additionally, deploying this application on a web server or cloud platform is necessary for real-world usage.