Chatbot using python phase 5

Introduction

Chatbots are computer programs designed to simulate human conversation and interact with users through text or voice. They are used in various applications, including customer support, information retrieval, and entertainment. Python is a popular programming language for developing chatbots due to its simplicity and rich ecosystem of libraries.

Here's a high-level introduction to building a chatbot using Python:

1. **Choose a Framework or Library:**
   - Python offers several libraries and frameworks for chatbot development. Some popular ones include ChatterBot, NLTK, spaCy, and Rasa. Choose the one that best fits your project's requirements.

2. **Define Your Bot's Purpose**:
   - Determine the purpose of your chatbot. Is it for customer support, providing information, or just for fun? Understanding your bot's goal will help shape its design and functionality.

3. **Data Collection and Preprocessing**:
   - To train your chatbot, you'll need a dataset of conversations or questions and answers. Collect and preprocess this data to make it suitable for training your chatbot's model.

4. **Natural Language Processing (NLP):**
   - NLP is a critical component of chatbots. It allows your bot to understand user input, extract relevant information, and generate appropriate responses. Many NLP libraries are available in Python to assist with these tasks.

5. **Build and Train the Model:**
   - Use the chosen library or framework to build your chatbot's model. This typically involves training the model on your dataset to understand and respond to user queries effectively.

6. **User Interaction**:
   - Implement the user interface for your chatbot. This could be a web-based chat interface, a messaging app integration, or a command-line interface.

7. **Continuous Improvement:**
   - Chatbots can be improved over time by analyzing user interactions and feedback. You can update the bot's responses, add new features, and enhance its capabilities.

8. **Deployment**:

- Once your chatbot is ready, deploy it to a platform where users can interact with it. This might involve setting up a web server, integrating with messaging platforms, or embedding it in your website or application.

9. **Testing and Maintenance**:
   - Regularly test your chatbot to identify and fix any issues or bugs. Maintenance is essential to keep the chatbot running smoothly.

10. **Scaling**:
    - If your chatbot gains popularity, you may need to scale it to handle a larger user base. This can involve optimizing performance and adding more resources.

Python offers a wide range of tools and libraries for chatbot development, making it a versatile choice for creating conversational agents. Depending on your project's complexity, you can create simple rule-based chatbots or more advanced machine learning-based chatbots using Python.

Tools and software used in chatbot

Chatbots are computer programs designed to simulate human conversation and interact with users through text or voice. They are used in various applications, including customer support, information retrieval, and entertainment. Python is a popular programming language for developing chatbots due to its simplicity and rich ecosystem of libraries.

Here's a high-level introduction to building a chatbot using Python:

1. **Choose a Framework or Library**:
   - Python offers several libraries and frameworks for chatbot development. Some popular ones include ChatterBot, NLTK, spaCy, and Rasa. Choose the one that best fits your project's requirements.

2. **Define Your Bot's Purpose:**
   - Determine the purpose of your chatbot. Is it for customer support, providing information, or just for fun? Understanding your bot's goal will help shape its design and functionality.

3. **Data Collection and Preprocessing**:
   - To train your chatbot, you'll need a dataset of conversations or questions and answers. Collect and preprocess this data to make it suitable for training your chatbot's model.

4. **Natural Language Processing (NLP):**
   - NLP is a critical component of chatbots. It allows your bot to understand user input, extract relevant information, and generate appropriate responses. Many NLP libraries are available in Python to assist with these tasks.

5. **Build and Train the Model:**
   - Use the chosen library or framework to build your chatbot's model. This typically involves training the model on your dataset to understand and respond to user queries effectively.

6. **User Interaction**:

- Implement the user interface for your chatbot. This could be a web-based chat interface, a messaging app integration, or a command-line interface.

7. **Continuous Improvement**:
   - Chatbots can be improved over time by analyzing user interactions and feedback. You can update the bot's responses, add new features, and enhance its capabilities.

8. **Deployment**:
   - Once your chatbot is ready, deploy it to a platform where users can interact with it. This might involve setting up a web server, integrating with messaging platforms, or embedding it in your website or application.

9. **Testing and Maintenance**:
   - Regularly test your chatbot to identify and fix any issues or bugs. Maintenance is essential to keep the chatbot running smoothly.

10. **Scaling**:
    - If your chatbot gains popularity, you may need to scale it to handle a larger user base. This can involve optimizing performance and adding more resources.

Python offers a wide range of tools and libraries for chatbot development, making it a versatile choice for creating conversational agents. Depending on your project's complexity, you can create simple rule-based chatbots or more advanced machine learning-based chatbots using Python.

Problem Definition and Design Thinking

Problem statement:
To define a problem for a chatbot in Python, you'll need to outline the following key elements:
1. **Purpose**: Clearly state the purpose of your chatbot. What problem is it designed to solve or what tasks should it assist with?
2. **Target Audience**: Describe the intended users or audience for your chatbot. Who will be interacting with it, and what are
their expectations?
3. **Functional Requirements**: List the specific functions or tasks your chatbot should perform. For example, if it's a customer
support chatbot, it should be able to answer common customer queries.
4. **Non-Functional Requirements**: Specify any non-functional requirements, such as performance, scalability, security, and
user experience considerations.
5. **Data Sources**: Identify the sources of data your chatbot will rely on. This could include databases, APIs, or other external
data repositories.
6. **Dialogue Flow**: Outline the conversation flow or dialog structure of your chatbot. Define the possible user inputs and the
corresponding bot responses.
7. **NLU (Natural Language Understanding):** Explain how your chatbot will understand user input. Will it use pre-trained NLP
models, custom-built NLU components, or a combination of both?

8.:**Integration**: Specify any integrations with external systems or services that your chatbot requires to perform its tasks.
Here's a simple example of a problem definition for a basic chatbot:
```
**Chatbot Problem Definition**
1. **Purpose:** To provide travel information and booking assistance for users planning vacations.
2. **Target Audience:** Travel enthusiasts looking for destination information, flight bookings, and hotel reservations.
3. **Functional Requirements:**
- Provide information about popular travel destinations.
- Assist users in finding flights based on their preferences.
- Help users book hotels at their chosen destination.
- Answer common travel-related questions.
4. **Non-Functional Requirements:**
- Response time should be under 2 seconds.
- The chatbot should be able to handle at least 100 concurrent users.
- User data should be securely handled and stored.
5. **Data Sources:**
- Flight data from a third-party flight booking API.
- Hotel information from a hotel booking service API.
- User profiles and preferences stored in a database.
6. **Dialogue Flow:** Define the conversation flow with sample user inputs and bot responses.
7. **NLU:** Utilize a pre-trained NLP model to understand user queries.
8. **Integration:** Integrate with the flight booking API and hotel booking service API for real-time bookings.
```
This problem definition serves as a foundation for developing your chatbot in Python. It helps you clarify the chatbot's purpose
and requirements before diving into the implementation
Data set link:
Data set link:https://www.kaggle.com/datasets/grafstor/simple-dialogs-for-chatbot
Problem definition:
The problem definition of a chatbot in Python typically involves creating a program that can interact with users in a conversational manner. Here's a basic outline of the problem definition:
Problem Statement: Develop a chatbot in Python that can engage in text-based conversations with users, providing information, answering questions, and assisting with tasks.
Key Requirements:
1. User Input: The chatbot should be able to accept user input in the form of text messages or questions.
2. Natural Language Understanding: The chatbot should be able to understand the natural language used by users, including recognizing intents and entities in the input.
3. Response Generation: The chatbot should generate appropriate responses to user queries or statements. These responses should be contextually relevant and grammatically correct.
4. Information Retrieval: If the chatbot is designed to provide information, it should be able to retrieve data from a database, external APIs, or other sources.
5 Conversation Management: The chatbot should manage the flow of the conversation, keeping track

of context and maintaining a coherent dialogue.
6. Error Handling: The chatbot should gracefully handle situations where it cannot understand the user's input or cannot fulfill a request.
7. Integration: Depending on the use case, the chatbot may need to integrate with external services, databases, or systems.
Approach:
1. Natural Language Processing (NLP): Implement NLP techniques to understand and process user input. Libraries like NLTK, spaCy, or Hugging Face Transformers can be helpful.
2. Dialog Management: Implement a dialog management system to keep track of the conversation context and generate appropriate responses.
3. Data Retrieval: If the chatbot requires access to external data sources, integrate APIs or databases to retrieve relevant information.
4. Response Generation: Use NLP techniques to generate responses that are contextually appropriate and user-friendly.
5. User Interface: Develop a user interface, which can be a command-line interface or a web-based chat interface, to interact with the chatbot.
6. Testing and Evaluation: Thoroughly test the chatbot with various user inputs to ensure it performs as expected and refine its responses based on user feedback.
7. Deployment: Deploy the chatbot in a suitable environment, which could be a web server, cloud platform, or an application.
Evaluation Metrics:
The performance of the chatbot can be evaluated using metrics such as accuracy in understanding user intent, response coherence, user satisfaction (gathered through user feedback), and the ability to handle a wide range of user inputs effectively.
Remember that the specific problem definition and requirements may vary depending on the chatbot's intended use case, whether it's for customer support, information retrieval, virtual assistance, or any other purpose.

Design thinking:

Design thinking is a problem-solving approach that focuses on understanding user needs and iteratively designing solutions to meet those needs. When applying design thinking to chatbot development in Python, you can follow these steps:
1. Empathize:
- Identify the target audience for your chatbot.
- Conduct user research to understand their needs, pain points, and preferences.
2. Define:
- Clearly define the problem your chatbot will solve.
- Create user personas and user stories to guide development.
3. Ideate:
- Brainstorm and generate ideas for your chatbot's features and functionality.
- Consider different Python libraries or frameworks for chatbot development, such as NLTK, spaCy, or Rasa.
4. Prototype:
- Create a basic prototype of your chatbot's conversation flow using pseudocode or a flowchart.
- Choose a Python chatbot development framework and start building a minimal viable product (MVP).
5. Test:
- Gather feedback from potential users and stakeholders on your chatbot's prototype.
- Use Python testing frameworks like pytest to identify and fix bugs.
6. Iterate:

- Refine your chatbot's design based on user feedback.
- Continue to improve its functionality and user experience.7. 7.Implement:
- Develop the full chatbot using Python and integrate it with any necessary APIs or databases.
- Ensure the chatbot's code is clean, maintainable, and well-documented.
8. Deploy:
- Host your chatbot on a server or cloud platform so users can access it.
- Set up continuous integration and deployment (CI/CD) pipelines if needed.
9. Monitor:
- Implement analytics and logging to track user interactions and identify areas for improvement.
- Use Python libraries like Matplotlib or Pandas for data analysis.
10. Refine:
- Regularly review chatbot performance and user feedback to make ongoing improvements.
- Add new features or capabilities as needed.

Throughout the design thinking process, you'll likely use various Python libraries and frameworks for natural language processing (NLP), user interface design, and backend development, depending on your chatbot's requirements. Additionally, consider user-centered design principles to create a chatbot that provides value and a positive user experience

Design into Innovation

Chatbot in Python:

According to this phase we are going to do about what are the innovations that we are required to innovate python in a new type and use the Chatbot in an innovative way.

Here now the Chatbot is instructed to done a answer which was asked by the user but the chatbot are much easier to design an algorithm to install the chatbot program.In this phase we innovate the Chatbot which we are going to integrate it with a Website or an Application.By integrate it with such a Website or an Application we are making that the chatbot to work as an external software model. The Chatbot which we are programmed as in Python we could run it in Python but by integrating it to any other software such as Website or Application we can use it anywhere from the app or the website Url. Here we all are know about CHATGPT it is an Chatbot but we can access the Chatbot in either external application or website. The Chatbots are mostly used in many fields in various roles and areas.

The world have already witnessed many innovative chatbots in the modern days ,Some Chatbots are

COVIDAsha bot
Endurance bot
Casper bot
UNICEF bot

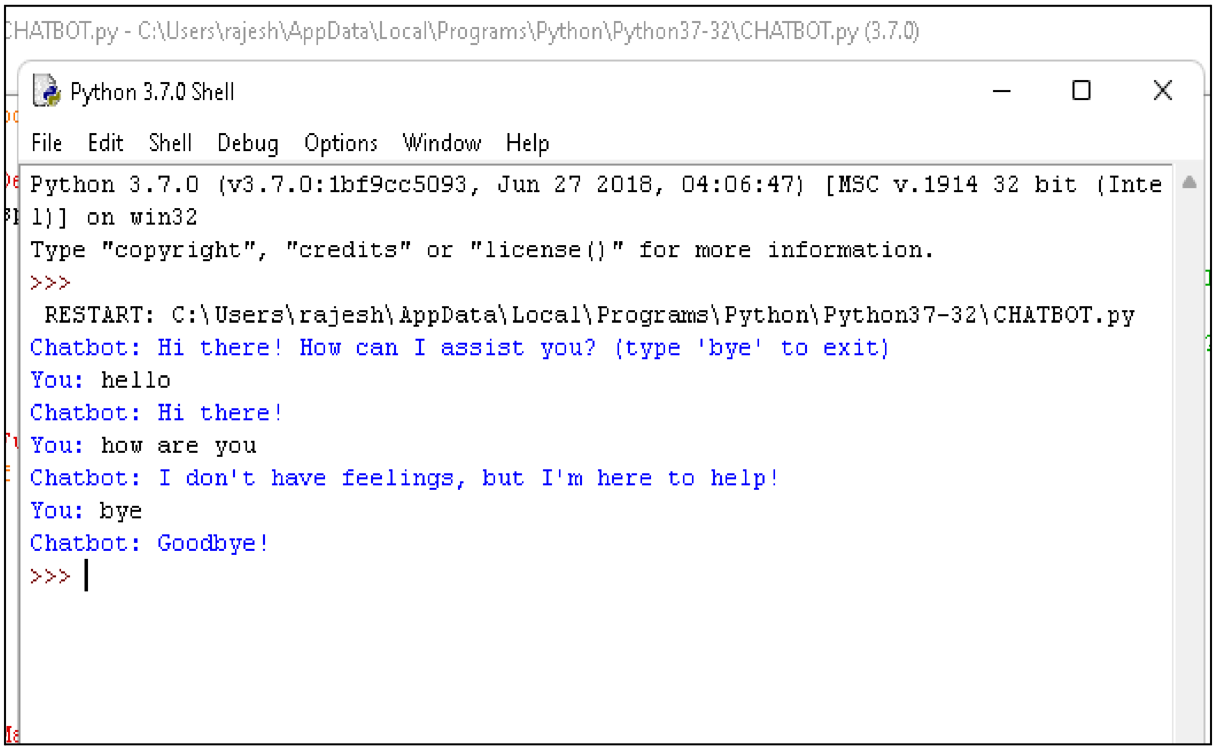
Here from the datasets that are already given the Python program to Develop or Run a Python Chatbot ,we were added or attended a small set of datasets we are created and here the Python Program and the Sample output in the python Software to be added.

After the Submission the Chatbot Python program to be integrated to a Website or Application then the Final Submission to be Submitted with model that we are created.

SAMPLE PYTHON PROGRAM OF CHATBOT:

```
import random
# Define responses
responses = {
            "hello": ["Hi there!", "Hello!", "Hey!"],
                    "how are you": ["I'm just a bot, but I'm doing fine.", "I don't have feelings, but I'm here to
help!"],
                     "bye": ["Goodbye!", "See you later!", "Bye bye!"],
                    "default": ["I'm not sure how to respond to that.", "Could you please rephrase that?", "I'm
still learning!"]
}
# Function to get a response
 def get_response(message):
                    message = message.lower()
                    if message in responses:
                              return random.choice(responses[message])
                    else:
                               return random.choice(responses["default"])
 # Main loop
 print("Chatbot: Hi there! How can I assist you? (type 'bye' to exit)")
while True:
          user_input = input("You: ")
          if user_input.lower() == "bye":
                    print("Chatbot: Goodbye!")
                    break
         response = get_response(user_input)
         print("Chatbot:", response)
```

SAMPLE OUTPUT FOR THE PYTHON PROGRAM:


CHATBOT.py - C:\Users\rajesh\AppData\Local\Programs\Python\Python37-32\CHATBOT.py (3.7.0)
Python 3.7.0 Shell
File Edit Shell Debug Options Window Help

```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Inte
l)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
 RESTART: C:\Users\rajesh\AppData\Local\Programs\Python\Python37-32\CHATBOT.py
Chatbot: Hi there! How can I assist you? (type 'bye' to exit)
You: hello
Chatbot: Hi there!
You: how are you
Chatbot: I don't have feelings, but I'm here to help!
You: bye
Chatbot: Goodbye!
>>> 
```

Building loading and preprocessing the data setChatbot in Python and Flask

Natural Language Processing or NLP is a prerequisite for our project. NLP allows computers and algorithms to understand human interactions via various languages. In order to process a large amount of natural language data, an AI will definitely need NLP or Natural Language Processing. Currently, we have a number of NLP research ongoing in order to improve the AI chatbots and help them understand the complicated nuances and undertones of human conversations.

Chatbots are nothing but applications that are used by businesses or other entities to conduct an automatic conversation between a human and an AI. These conversations may be via text or speech. Chatbots are required to understand and mimic human conversation while interacting with humans from all over the world. From the first chatbot to be created ELIZA to Amazon's ALEXA today, chatbots have come a long way. In this tutorial, we are going to cover all the basics you need to follow along and create a basic chatbot that can understand human interaction and also respond accordingly. We will be using speech recognition APIs and also pre-trained Transformer models.

NLP

NLP stands for Natural Language Processing. Using NLP technology, you can help a machine understand human speech and spoken words. NLP combines computational linguistics that is the rule-based modelling of the human spoken language with intelligent algorithms such as statistical, machine, and deep learning algorithms. These technologies together create the smart voice assistants and chatbots that you may be used in everyday life.

There are a number of human errors, differences, and special intonations that humans use every day in their speech. NLP technology allows the machine to understand, process, and respond to large volumes of text rapidly in real-time. In everyday life, you have encountered NLP tech in voice-guided GPS apps, virtual assistants, speech-to-text note creation apps, and other app support chatbots. This tech has found immense use cases in the business sphere where it's used to streamline processes, monitor employee productivity, and increase sales and after-sales efficiency.

TYPES OF CHATBOTS

Chatbots are a relatively recent concept and despite having a huge number of programs and NLP tools, we basically have just two different categories of chatbots based on the NLP technology that they utilize. These two types of chatbots are as follows:

Scripted chatbots: Scripted chatbots are classified as chatbots that work on pre-determined scripts that are created and stored in their library. Whenever a user types a query or speaks a query (in the case of chatbots equipped with speech to text conversion modules), the chatbot responds to this query according to the pre-determined script that is stored within its library.One of the cons of such a chatbot is the fact that user needs to provide their que3ry in a very structured manner with comma-separated commands or other forms of a regular expression that makes it easier for the bot to perform string analysis and understand the query.

Artificially Intelligent Chatbots: Artificially intelligent chatbots, as the name suggests, are created to mimic human-like traits and responses. NLP or Natural Language Processing is hugely responsible for enabling such chatbots to understand the dialects and undertones of human conversation. NLP combined with artificial intelligence creates a truly intelligent chatbot that can respond to nuanced questions and learn from every interaction to create better-suited responses the next time.

DEVELOPMENT

Importing libraries

```
import tensorflow as tf
from sklearn.model_selection import train_test_split



import unicodedata
import re
import numpy as np



import warnings
warnings.filterwarnings('ignore')
```

Data preprocessing

The basic text processing in NLP are:

Sentence Segmentation

Normalization

Tokenization

Segmentation

```
data=open('/content/dialogs.txt','r').read()

QA_list=[QA.split('\t') for QA in data.split('\n')]
print(QA_list[:5])
```

Output:

```
[['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."]]
```

```
questions=[row[0] for row in QA_list]

answers=[row[1] for row in QA_list]


print(questions[0:5])

print(answers[0:5])
```

Output:

```
['hi, how are you doing?', "i'm fine. how about yourself?", "i'm pretty good. thanks for asking.", 'no problem. so how have you been?', "i've been great. what about you?"]

["i'm fine. how about yourself?", "i'm pretty good. thanks for asking.", 'no problem. so how have you been?', "i've been great. what about you?", "i've been good. i'm in school right now."]
```

Normalization

```
def remove_diacritic(text):
    return ''.join(char for char in unicodedata.normalize('NFD',text)
                if unicodedata.category(char) !='Mn')


def preprocessing(text):



    text=remove_diacritic(text.lower().strip())



    text=re.sub(r"([?.!,¿])", r" \1 ", text)


    text= re.sub(r'[" "]+', " ", text)
    text=re.sub(r"[^a-zA-Z?.!,¿]+", " ", text)

    text=text.strip()

    text='<start> ' + text + ' <end>'

    return text

preprocessed_questions=[preprocessing(sen) for sen in questions]
preprocessed_answers=[preprocessing(sen) for sen in answers]



print(preprocessed_questions[0])
print(preprocessed_answers[0])
```

Output:

```
<start> hi , how are you doing ? <end>
<start> i m fine . how about yourself ? <end>
```

Tokenization

```
def tokenize(lang):
    lang_tokenizer = tf.keras.preprocessing.text.Tokenizer(
      filters='')

    lang_tokenizer.fit_on_texts(lang)

    return lang_tokenizer
```

Word Embedding

```
def vectorization(lang_tokenizer,lang):


    tensor = lang_tokenizer.texts_to_sequences(lang)


    tensor = tf.keras.preprocessing.sequence.pad_sequences(tensor,
                                                           padding='post')


    return tensor
```

Creating Dataset

```
def load_Dataset(data,size=None):

    if(size!=None):
        y,X=data[:size]
    else:
        y,X=data

    X_tokenizer=tokenize(X)
    y_tokenizer=tokenize(y)

    X_tensor=vectorization(X_tokenizer,X)
    y_tensor=vectorization(y_tokenizer,y)

    return  X_tensor,X_tokenizer, y_tensor, y_tokenizer


size=30000
```

```
data=preprocessed_answers,preprocessed_questions\

X_tensor,X_tokenizer, y_tensor, y_tokenizer=load_Dataset(data,size)

max_length_y, max_length_X = y_tensor.shape[1], X_tensor.shape[1]
```

Splitting Data

```
X_train, X_val, y_train, y_val = train_test_split(X_tensor, y_tensor, test_size=0.2)

print(len(X_train), len(y_train), len(X_val), len(y_val))
```

Output:

```
2980 2980 745 745
```

Tensorflow Dataset

```
BUFFER_SIZE = len(X_train)
BATCH_SIZE = 64
steps_per_epoch = len(X_train)//BATCH_SIZE
embedding_dim = 256
units = 1024
vocab_inp_size = len(X_tokenizer.word_index)+1
vocab_tar_size = len(y_tokenizer.word_index)+1

dataset = tf.data.Dataset.from_tensor_slices((X_train, y_train)).shuffle(BUFFER_SIZE)
dataset = dataset.batch(BATCH_SIZE, drop_remainder=True)



example_input_batch, example_target_batch = next(iter(dataset))
example_input_batch.shape, example_target_batch.shape
```

Output:

```
(TensorShape([64, 24]), TensorShape([64, 24]))
```

Model

Buliding Model Architecture

Encoder

```
class Encoder(tf.keras.Model):
  def __init__(self, vocab_size, embedding_dim, enc_units, batch_sz):
    super(Encoder, self).__init__()
    self.batch_sz = batch_sz
    self.enc_units = enc_units
    self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
    self.gru = tf.keras.layers.GRU(self.enc_units,
                                   return_sequences=True,
                                   return_state=True,
                                   recurrent_initializer='glorot_uniform')


  def call(self, x, hidden):
    x = self.embedding(x)
    output, state = self.gru(x, initial_state = hidden)
```

```
        return output, state



    def initialize_hidden_state(self):
        return tf.zeros((self.batch_sz, self.enc_units))

encoder = Encoder(vocab_inp_size, embedding_dim, units, BATCH_SIZE)



sample_hidden = encoder.initialize_hidden_state()
sample_output, sample_hidden = encoder(example_input_batch, sample_hidden)
print ('Encoder output shape: (batch size, sequence length, units) {}'.format(sample_output.shape))
print ('Encoder Hidden state shape: (batch size, units) {}'.format(sample_hidden.shape))
```

Output:

```
Encoder output shape: (batch size, sequence length, units) (64, 24, 1024)
Encoder Hidden state shape: (batch size, units) (64, 1024)
```

Attention Mechanism

```
class BahdanauAttention(tf.keras.layers.Layer):
    def __init__(self, units):
        super(BahdanauAttention, self).__init__()
        self.W1 = tf.keras.layers.Dense(units)
        self.W2 = tf.keras.layers.Dense(units)
        self.V = tf.keras.layers.Dense(1)



    def call(self, query, values):
        query_with_time_axis = tf.expand_dims(query, 1)



        score = self.V(tf.nn.tanh(
            self.W1(query_with_time_axis) + self.W2(values)))



        attention_weights = tf.nn.softmax(score, axis=1)


        context_vector = attention_weights * values
        context_vector = tf.reduce_sum(context_vector, axis=1)


        return context_vector, attention_weights

attention_layer = BahdanauAttention(10)
attention_result, attention_weights = attention_layer(sample_hidden, sample_output)


print("Attention result shape: (batch size, units) {}".format(attention_result.shape))
print("Attention weights shape: (batch_size, sequence_length, 1) {}".format(attention_weights.shape))
```

Output:

```
Attention result shape: (batch size, units) (64, 1024)
Attention weights shape: (batch_size, sequence_length, 1) (64, 24, 1)
```

Decoder

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

```
def __init__(self, vocab_size, embedding_dim, dec_units, batch_sz):
  super(Decoder, self).__init__()
  self.batch_sz = batch_sz
  self.dec_units = dec_units
  self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
  self.gru = tf.keras.layers.GRU(self.dec_units,
                                 return_sequences=True,
                                 return_state=True,
                                 recurrent_initializer='glorot_uniform')
  self.fc = tf.keras.layers.Dense(vocab_size)



  self.attention = BahdanauAttention(self.dec_units)


def call(self, x, hidden, enc_output):

  context_vector, attention_weights = self.attention(hidden, enc_output)


  x = self.embedding(x)



  x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1)



  output, state = self.gru(x)
```

```
        output = tf.reshape(output, (-1, output.shape[2]))



        x = self.fc(output)



        return x, state, attention_weights
decoder = Decoder(vocab_tar_size, embedding_dim, units, BATCH_SIZE)



sample_decoder_output, _, _ = decoder(tf.random.uniform((BATCH_SIZE, 1)),
                                      sample_hidden, sample_output)



print ('Decoder output shape: (batch_size, vocab size) {}'.format(sample_decoder_output.shape))
```

Output:

```
Decoder output shape: (batch_size, vocab size) (64, 2349)
```

Training Model

```
optimizer = tf.keras.optimizers.Adam()
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')



def loss_function(real, pred):
    mask = tf.math.logical_not(tf.math.equal(real, 0))
    loss_ = loss_object(real, pred)
    mask = tf.cast(mask, dtype=loss_.dtype)
    loss_ *= mask

    return tf.reduce_mean(loss_)

@tf.function
def train_step(inp, targ, enc_hidden):
    loss = 0

    with tf.GradientTape() as tape:
        enc_output, enc_hidden = encoder(inp, enc_hidden)

        dec_hidden = enc_hidden

        dec_input = tf.expand_dims([y_tokenizer.word_index['<start>']] * BATCH_SIZE, 1)

        for t in range(1, targ.shape[1]):
            predictions, dec_hidden, _ = decoder(dec_input, dec_hidden, enc_output)

            loss += loss_function(targ[:, t], predictions)
            dec_input = tf.expand_dims(targ[:, t], 1)


    batch_loss = (loss / int(targ.shape[1]))


    variables = encoder.trainable_variables + decoder.trainable_variables


    gradients = tape.gradient(loss, variables)


    optimizer.apply_gradients(zip(gradients, variables))


    return batch_loss

EPOCHS = 40


for epoch in range(1, EPOCHS + 1):
    enc_hidden = encoder.initialize_hidden_state()
    total_loss = 0


    for (batch, (inp, targ)) in enumerate(dataset.take(steps_per_epoch)):
        batch_loss = train_step(inp, targ, enc_hidden)
        total_loss += batch_loss
    if(epoch % 4 == 0):
        print('Epoch:{:3d} Loss:{:.4f}'.format(epoch,
                                          total_loss / steps_per_epoch))
```

Output:

```
 Epoch:  4 Loss:1.5338
 Epoch:  8 Loss:1.2803
 Epoch: 12 Loss:1.0975
 Epoch: 16 Loss:0.9404
 Epoch: 20 Loss:0.7773
 Epoch: 24 Loss:0.6040
 Epoch: 28 Loss:0.4042
 Epoch: 32 Loss:0.2233
 Epoch: 36 Loss:0.0989
 Epoch: 40 Loss:0.0470
```

Model Evaluation

```
def remove_tags(sentence):
    return sentence.split("<start>")[-1].split("<end>")[0]


def evaluate(sentence):
    sentence = preprocessing(sentence)



    inputs = [X_tokenizer.word_index[i] for i in sentence.split(' ')]
    inputs = tf.keras.preprocessing.sequence.pad_sequences([inputs],
                                                           maxlen=max_length_X,
                                                           padding='post')
    inputs = tf.convert_to_tensor(inputs)
```

```
result = ''


hidden = [tf.zeros((1, units))]
enc_out, enc_hidden = encoder(inputs, hidden)


dec_hidden = enc_hidden
dec_input = tf.expand_dims([y_tokenizer.word_index['<start>']], 0)


for t in range(max_length_y):
  predictions, dec_hidden, attention_weights = decoder(dec_input,
                                                       dec_hidden,
                                                       enc_out)


  attention_weights = tf.reshape(attention_weights, (-1, ))


  predicted_id = tf.argmax(predictions[0]).numpy()


  result += y_tokenizer.index_word[predicted_id] + ' '


  if y_tokenizer.index_word[predicted_id] == '<end>':
    return remove_tags(result), remove_tags(sentence)
```

```
        dec_input = tf.expand_dims([predicted_id], 0)

    return remove_tags(result), remove_tags(sentence)

def ask(sentence):
    result, sentence = evaluate(sentence)

    print('Question: %s' % (sentence))
    print('Predicted answer: {}'.format(result))

ask(questions[1])
```

Output:

```
Question:  i m fine . how about yourself ?
Predicted answer: I m pretty good . thanks for asking
```

Conclusion

A chatbot in Python is a powerful and versatile tool that can be used for a wide range of applications. In conclusion, here are some key points to consider:

1. Accessibility: Python is a popular and accessible programming language, making it a great choice for developing chatbots. It offers a wide range of libraries and frameworks that simplify the development process.

2. Natural Language Processing (NLP): Python boasts robust NLP libraries such as NLTK, spaCy, and the Hugging Face Transformers library, which enable chatbots to understand and generate human-like text. These libraries have made it easier than ever to create sophisticated conversational agents.

3. Versatility: Python chatbots can be implemented in various domains, from customer service and e-commerce to healthcare and education. Their versatility allows them to adapt to a wide range of industries and tasks.

4. Integration: Python chatbots can easily integrate with existing systems and platforms, allowing for seamless communication with users through popular messaging services like WhatsApp, Facebook Messenger, or Slack.

5. Machine Learning: Python's ecosystem is rich with machine learning tools, such as scikit-learn and TensorFlow, which can be used to enhance chatbot functionality through supervised or reinforcement learning, improving its ability to understand user intent and generate relevant responses.

6. Continuous Improvement: Chatbots can learn and adapt over time through user interactions and feedback. By implementing data collection and analysis, Python chatbots can be continuously improved to provide better user experiences.

7. Scalability: Python chatbots can be deployed on a variety of platforms, from web applications to mobile apps, and they can scale to handle a large number of users simultaneously, making them suitable for businesses of all sizes.

8. Ethical Considerations: It's important to consider ethical and privacy aspects when developing chatbots. Developers should be mindful of user data and privacy concerns, and ensure that chatbots provide value while respecting ethical guidelines.

In conclusion, Python is an excellent choice for building chatbots due to its accessibility, powerful NLP libraries, versatility, and robust ecosystem. With the right design, implementation, and ongoing improvements, Python chatbots can provide efficient and engaging interactions with users in a wide range of domains.

Chatbot in Python and Flask

What is a Chatbot?

A Chatbot, also called an Artificial chat agent, is a software program driven by machine learning algorithms that aim at simulating a human-human like conversation with a user by either taking input as text or speech from the user.

Where is it used?

Chatbots have extensive usage, and we can not expound on all the possibilities where it can be of use. But basically, you'll find them in: Help desks, transaction processing, customer support, booking services, and providing 24-7 real-time chat with clients.

Do I need one?

Well, this is a personalized opinion where one has to do a cost-benefit analysis and decide whether it is a worthwhile project.

At the current technology stand, most companies are slowly transitioning to use chatbots for their in-demand day-day services. A good example that everybody uses is the Google Assistant, Apple Siri, Samsung Bixby, and Amazon Alexa.

In this article, we will learn how to create one in Python using TensorFlow to train the model and Natural Language Processing(nltk) to help the machine understand user queries.

There are two broad categories of chatbots:

Rule-Based approach - Here the bot is trained based on some set rules. It is from these rules that the bot can process simple queries but can fail to process complex ones.

Self-Learning approach - Here the bot uses some machine learning algorithms and techniques to chat. It is further subcategorized into two:

Retrieval-Based models - In this model, the bot retrieves the best response from a list depending on the user input.

Generative models - This model comes up with an answer rather than searching from a given list. These are the Intelligent Bots.

In this chatbot, we will use the rule-based approach.

Terms to encounter

Natural Language Processing(nltk) - This is a subfield of linguistics, computer science, information engineering, and artificial intelligence concerned with the interactions between computers and human (natural) languages.

Lemmatization - This is the process of grouping together the different inflected forms of a word so they can be analyzed as a single item and is a variation of stemming. For example “feet” and “foot” are both recognized as “foot”.

Stemming - This is the process of reducing inflected words to their word stem, base, or root form. For example, if we were to stem the word “eat”, “eating”, “eats”, the result would be the single word “eat”.

Tokenization - Tokens are individual words and “tokenization” is taking a text or set of text and breaking it up into its individual words or sentences.

Bag of Words - This is an NLP technique of text modeling for representing text data for machine learning algorithms. It is a way of extracting features from the text for use in machine learning algorithms.

Let's start programming now

Directory structure

```
|-- Static
|           |-- style.css
|-- Templates
|           |-- index.html
|-- app.py
|-- train.py
|-- intents.json
|-- words.pkl
|-- classes.pkl
|-- chatbot_model.h5
```

First, we need to make sure that we have all the required libraries and modules. Install tensorflow, nltk, and flask using the following command. pip install tensorflow

```
pip install tensorflow-gpu pip install nltk
pip install Flask
```

Our intents.json file should look something like this: Here I'm sorry it's quite a long file I can't embed it here.

Now that we have everything ready, Let's start by creating the train.py file. We need to import all the packages and libraries that we're going to be using.

```
# libraries import random
from tensorflow.keras.optimizers import SGD from keras.layers import Dense, Dropout from keras.models import load_model
from keras.models import Sequential import numpy as np
import pickle import json import nltk
```

```
from nltk.stem import WordNetLemmatizer
```

Download punkt, omw-1.4, and wordnet package. Punkt is a pre-trained tokenizer model for the English language that divides the text into a list of sentences.

```
nltk.download('omw-1.4') nltk.download("punkt") nltk.download("wordnet")
```

We now need to initialize some files and load our training data. Note that we are going to be ignoring "?" and "!". If you have some other symbols or letters that you want the model to ignore you can add them at the ignore_words array.

```
# init file words = [] classes = [] documents = []

ignore_words = ["?", "!"]

data_file = open("intents.json").read() intents = json.loads(data_file)
```

Then we tokenize our words

```
for intent in intents["intents"]:

for pattern in intent["patterns"]:


# take each word and tokenize it w = nltk.word_tokenize(pattern) words.extend(w)

# adding documents documents.append((w, intent["tag"]))


# adding classes to our class list if intent["tag"] not in classes:

classes.append(intent["tag"])
```

Next, we shall lemmatize the words and dump them in a pickle file words = [lemmatizer.lemmatize(w.lower()) for w in words if w not in ignore_words] words = sorted(list(set(words)))

```
classes = sorted(list(set(classes))) print(len(documents), "documents") print(len(classes), "classes", classes) print(len(words), "unique lemmatized words", words)


pickle.dump(words, open("words.pkl", "wb")) pickle.dump(classes, open("classes.pkl", "wb"))
```

Now that we have our training data in place, we initialize the model training

# initializing training data training = []

output_empty = [0] * len(classes) for doc in documents:

# initializing bag of words bag = []

# list of tokenized words for the pattern pattern_words = doc[0]

# lemmatize each word - create base word, in attempt to represent related words pattern_words = [lemmatizer.lemmatize(word.lower()) for word in pattern_words] # create our bag of words array with 1, if word match found in current pattern

for w in words:

bag.append(1) if w in pattern_words else bag.append(0)

# output is a '0' for each tag and '1' for current tag (for each pattern) output_row = list(output_empty)

output_row[classes.index(doc[1])] = 1

training.append([bag, output_row])

# shuffle our features and turn into np.array random.shuffle(training)

training = np.array(training)

# create train and test lists. X - patterns, Y - intents train_x = list(training[:, 0])

train_y = list(training[:, 1]) print("Training data created")

We shall create a 3-layer output model. The first layer having 128 neurons, the second layer having 64 neurons, and the third layer contains the number of neurons equal to the number of intents to predict output intent with softmax. We shall be using ReLu activation function as it's easier to train and achieves good perfomance.

model = Sequential()

model.add(Dense(128, input_shape=(len(train_x[0]),), activation="relu")) model.add(Dropout(0.5))

model.add(Dense(64, activation="relu")) model.add(Dropout(0.5)) model.add(Dense(len(train_y[0]), activation="softmax")) model.summary()

Then, we compile the model. Stochastic gradient descent with Nesterov accelerated gradient gives good results for this model. I won't go into details about Stochastic gradient descent as this is a vastly complicated topic on its own.

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True) model.compile(loss="categorical_crossentropy", optimizer=sgd, metrics=["accuracy"])

OPTIONAL: For choosing an optimal number of training epochs to avoid underfitting or overfitting use an early stopping callback to Keras based on either accuracy or loss monitoring. If the loss is being monitored, training comes to halt when there is an increment observed in loss values. Or, If accuracy is being monitored, training comes to halt when there is a decrement observed in accuracy values.

```
from Keras import callbacks

earlystopping = callbacks.EarlyStopping(monitor ="loss", mode ="min", patience = 5, restore_best_weights = True)

callbacks =[earlystopping]
```

Now we can train our model and save it for fast access from the Flask REST API without the need of retraining.

```
hist = model.fit(np.array(train_x), np.array(train_y), epochs=200, batch_size=5, verbose=1)
model.save("chatbot_model.h5", hist)

print("model created")
```

Your train.py file now should look like this:

```
# libraries



import random



from tensorflow.keras.optimizers import SGD


from keras.layers import Dense, Dropout



from keras.models import load_model



from keras.models import Sequential


import numpy as np
import pickle

import json

import nltk

from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

nltk.download('omw-1.4')

nltk.download("punkt")

nltk.download("wordnet")
```

```
# init file

words = []

classes = []

documents = []

ignore_words = ["?", "!"]

data_file = open("intents.json").read()

intents = json.loads(data_file)

# words

for intent in intents["intents"]:

for pattern in intent["patterns"]:
```

```
# take each word and tokenize it

w = nltk.word_tokenize(pattern)

words.extend(w)

# adding documents

documents.append((w, intent["tag"]))

list
# adding classes to our class

if intent["tag"] not in classes:
```

```
classes.append(intent["tag"])

# lemmatizer

words = [lemmatizer.lemmatize(w.lower()) for w in words if w not in ignore_words]

words = sorted(list(set(words)))

classes = sorted(list(set(classes)))

print(len(documents), "documents")
print(len(classes), "classes", classes)

print(len(words), "unique lemmatized words", words)

pickle.dump(words, open("words.pkl", "wb"))

pickle.dump(classes, open("classes.pkl", "wb"))

# initializing training data
```

```
training = []

output_empty = [0] * len(classes)

for doc in documents:

# initializing bag of words

bag = []

# list of tokenized words for the pattern

pattern_words = doc[0]

# lemmatize each word - create base word, in attempt to represent related words

pattern_words = [lemmatizer.lemmatize(word.lower()) for word in pattern_words]

# create our bag of words array with 1, if word match found in current pattern

for w in words:

bag.append(1) if w in pattern_words else bag.append(0)
```

```
# output is a '0' for each tag and '1' for current tag (for each pattern)

output_row = list(output_empty)

output_row[classes.index(doc[1])] = 1

training.append([bag, output_row])

# shuffle our features and turn into np.array

random.shuffle(training)

training = np.array(training)

# create train and test lists. X - patterns, Y - intents

train_x = list(training[:, 0])

train_y = list(training[:, 1])
```

```
print("Training data created")

# actual training

model = Sequential()

model.add(Dense(128, input_shape=(len(train_x[0]),), activation="relu"))

model.add(Dropout(0.5))

model.add(Dense(64, activation="relu"))

model.add(Dropout(0.5))

model.add(Dense(len(train_y[0]), activation="softmax"))

model.summary()

# Compile model. Stochastic gradient descent with Nesterov accelerated gradient gives good results for this model
```

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)

model.compile(loss="categorical_crossent ropy", optimizer=sgd, metrics=["accuracy"])

#Optional snippet - I have commented it as I did not use it.

# from keras import callbacks

# earlystopping = callbacks.EarlyStopping(monitor ="loss", mode ="min", patience = 5, restore_best_weights = True)

# callbacks =[earlystopping]

# fitting and saving the model

```
hist = model.fit(np.array(train_x), np.array(train_y), epochs=200, batch_size=5, verbose=1)

model.save("chatbot_model.h5", hist)

print("model created")
```

Run the train.py to create the model.

Now that we are done with training let's create the Flask interface to initialize the chat functionalities.

We load the required libraries and initialize the Flask app

```
# libraries import random

import numpy as np import pickle

import json

from flask import Flask, render_template, request from flask_ngrok import run_with_ngrok

import nltk

from keras.models import load_model from nltk.stem import WordNetLemmatizer lemmatizer = WordNetLemmatizer()


# chat initialization

model = load_model("chatbot_model.h5") intents = json.loads(open("intents.json").read()) words = pickle.load(open("words.pkl", "rb")) classes = pickle.load(open("classes.pkl", "rb"))

app = Flask(  name  )

#run_with_ngrok(app) -Use this option if you have ngrok and you want to expose your chatbot to the real world

@app.route("/") def home():

return render_template("index.html")
```

This function will be called every time a user sends a message to the chatbot and returns a corresponding response based on the user query. @app.route("/get", methods=["POST"])

```
def chatbot_response():

msg = request.form["msg"]
```

```
if msg.startswith('my name is'):


name = msg[11:]

ints = predict_class(msg, model) res1 = getResponse(ints, intents) res =res1.replace("{n}",name)

elif msg.startswith('hi my name is'): name = msg[14:]

ints = predict_class(msg, model) res1 = getResponse(ints, intents) res =res1.replace("{n}",name)

else:

ints = predict_class(msg, model) res = getResponse(ints, intents)

return res
```

The above function will call the following functions which clean up sentences and return a bag of words based on the user input.

```
def clean_up_sentence(sentence): sentence_words = nltk.word_tokenize(sentence)

sentence_words = [lemmatizer.lemmatize(word.lower()) for word in sentence_words] return sentence_words



# return bag of words array: 0 or 1 for each word in the bag that exists in the sentence def bow(sentence, words, show_details=True):

# tokenize the pattern

sentence_words = clean_up_sentence(sentence)

# bag of words - matrix of N words, vocabulary matrix bag = [0] * len(words)

for s in sentence_words:

for i, w in enumerate(words): if w == s:

# assign 1 if current word is in the vocabulary position bag[i] = 1

if show_details:

print("found in bag: %s" % w) return np.array(bag)
```

The next functions are for predicting the response to give to the user where they fetch that response from the chatbot_model.h5 file generated after the training.

```
def predict_class(sentence, model):

# filter out predictions below a threshold

p = bow(sentence, words, show_details=False)
```

```
res = model.predict(np.array([p]))[0]

ERROR_THRESHOLD = 0.25

results = [[i, r] for i, r in enumerate(res) if r > ERROR_THRESHOLD] # sort by strength of probability

results.sort(key=lambda x: x[1], reverse=True) return_list = []

for r in results:

return_list.append({"intent": classes[r[0]], "probability": str(r[1])}) return return_list



def getResponse(ints, intents_json): tag = ints[0]["intent"]

list_of_intents = intents_json["intents"] for i in list_of_intents:

if i["tag"] == tag:

result = random.choice(i["responses"]) break

return result
```

Generally, our app.py should look like this:

```
# libraries



import random



import numpy as np



import pickle



import json
```

```
from flask import Flask, render_template, request

from flask_ngrok import run_with_ngrok


import nltk


from keras.models import load_model


from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()








# chat initialization


model = load_model("chatbot_model.h5")


intents = json.loads(open("intents.json").read())
```

```
words = pickle.load(open("words.pkl", "rb"))

classes = pickle.load(open("classes.pkl", "rb"))

app = Flask(  name  )

@app.route("/")

def home():

return render_template("index.html")
```

```
@app.route("/get", methods=["POST"])

def chatbot_response():


msg = request.form["msg"]


#checks is a user has given a name, in order to give a personalized feedback

if msg.startswith('my name is'):


name = msg[11:]


ints = predict_class(msg, model)


res1 = getResponse(ints, intents)


res =res1.replace("{n}",name)


elif msg.startswith('hi my name is'):


name = msg[14:]
```

```
ints = predict_class(msg, model)

res1 = getResponse(ints, intents)

res =res1.replace("{n}",name)

#if no name is passed execute normally
else:

ints = predict_class(msg, model)

res = getResponse(ints, intents)

return res

# chat functionalities
```

```
def clean_up_sentence(sentence):

sentence_words = nltk.word_tokenize(sentence)

sentence_words = [lemmatizer.lemmatize(word.lower()) for word in sentence_words]

return sentence_words

# return bag of words array: 0 or 1 for each word in the bag that exists in the sentence

def bow(sentence, words, show_details=True):

# tokenize the pattern

sentence_words = clean_up_sentence(sentence)

# bag of words - matrix of N words, vocabulary matrix

bag = [0] * len(words)
```

```
for s in sentence_words:

for i, w in enumerate(words):

if w == s:

# assign 1 if current word is in the vocabulary position

bag[i] = 1

if show_details:

print("found in bag:
%s" % w)

return np.array(bag)
```

```
def predict_class(sentence, model):



# filter out predictions below a threshold

p = bow(sentence, words, show_details=False)

res = model.predict(np.array([p]))[0]



ERROR_THRESHOLD = 0.25



results = [[i, r] for i, r in enumerate(res) if r > ERROR_THRESHOLD]

# sort by strength of probability

results.sort(key=lambda x: x[1], reverse=True)

return_list = []



for r in results:



return_list.append({"intent": classes[r[0]], "probability": str(r[1])})

return return_list
```

```
def getResponse(ints, intents_json):

tag = ints[0]["intent"]

list_of_intents = intents_json["intents"]

for i in list_of_intents:

if i["tag"] == tag:

result = random.choice(i["responses"])

break

return result
```

```
if   name == "  main  ":

app.run()
```

Lastly, we have the index.html file which I won't go into detail about as it's basic HTML and CSS.

```
<!DOCTYPE html>

<html>

<head>

<link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='style.css')}}" />

<!-- <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='css.css')}}"
/> -->

<script src="https://ajax.googleapis.com/ajax/li bs/jquery/3.2.1/jquery.min.js"></script>
```

```
</head>

<body>

<div class="row">

<div class="col-md-10 mr-auto ml-auto">

<h1>Dentrice AI ChatBot</h1>

<form>

<div id="chatbox">

<div class="col-md-8 ml-auto
mr-auto">

<p class="botText"><span>Hi! I'm Your bot.</span></p>

</div>

</div>

<div id="userInput" class="row">
```

```
<div class="col-md-10">

<input id="text" type="text" name="msg" placeholder="Message" class="form-control">

<button type="submit" id="send" class="btn
btn-warning">Send</button>

</div>

</div>

</form>

</div>

</div>

<script>
```

```
$(document).ready(function() {

$("form").on("submit", function(event) {

var rawText =
$("#text").val();

var userHtml = '<p class="userText"><span>' + rawText + "</span></p>";

$("#text").val("");

$("#chatbox").append(userHtml);

document.getElementById("userInput").scr ollIntoView({

block: "start",

behavior: "smooth",

});

$.ajax({
```

```
data: {

msg: rawText,

},

type: "POST",

url: "/get",

}).done(function(data) {

var botHtml = '<p class="botText"><span>' + data + "</span></p>";

$("#chatbox").append($.parseHTML(botHtml
));


document.getElementById("userInput").scr ollIntoView({

block: "start",
behavior: "smooth",

});

});

event.preventDefault();

});

});
</script>

</body>



</html>
```

Advantage of chatbot

Chatbots offer several advantages, including:

1. 24/7 Availability: Chatbots can provide instant responses and support around the clock, improving customer service and accessibility.

2. Cost-Efficiency: They reduce the need for human agents, saving on labor costs.

3. Scalability: Chatbots can handle multiple conversations simultaneously, making them scalable for businesses of all sizes.

4. Consistency: They provide consistent and accurate information, reducing human errors.

5. Quick Response Times: Chatbots can respond instantly, improving user experience.

6. Data Collection: They can collect valuable user data and insights for better decision-making.

7. Multilingual Support: Chatbots can communicate in multiple languages, catering to a global audience.

8. Automation: They automate repetitive tasks, freeing up human agents for more complex issues.

9. Integration: Chatbots can be integrated into various platforms, including websites, apps, and messaging apps.

10. Personalization: Advanced chatbots can offer personalized recommendations and content based on user preferences.

These advantages make chatbots a valuable tool for businesses and organizations across various industries.

Disadvantage of chatbot

One significant disadvantage of chatbots is that they can lack the ability to fully understand and engage in complex, nuanced, or emotionally sensitive conversations, as they rely on predefined algorithms and data. Additionally, they may struggle with context and can provide incorrect or irrelevant responses, frustrating users.

Conclusion

In conclusion, chatbots are computer programs designed to engage in natural language conversations with users. They have a wide range of applications, from customer support and virtual assistants to information retrieval and entertainment. Chatbots use various technologies like natural language processing and machine learning to understand and respond to user inputs. While they have made significant advancements in recent years, they are not without limitations, such as handling complex and context-aware conversations. However, as AI technology continues to evolve, chatbots are likely to play an increasingly important role in various industries and our daily lives.