CREATE A CHATBOT IN PYTHON

TEAM MEMBER

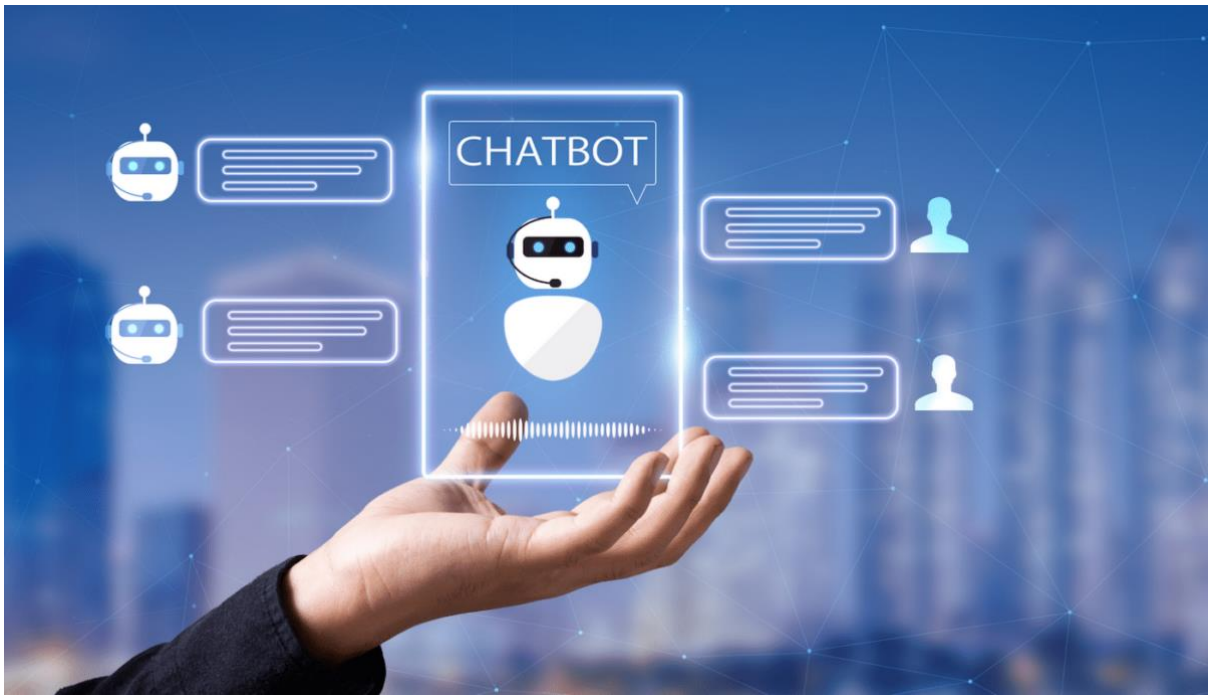
Demi Rithika.G 963321104018

Phase 5 submission document

Project Title: Create a Chatbot in python

Phase 5: Project Documentation & Submission

Topic: *In this section we will document the complete project and prepare it for submission.*



Create a Chatbot in python

Introduction:

Creating a chatbot in Python involves several steps and considerations. Here's an introduction to the process, summarized in three key points:

1. Understanding the Basics: To create a chatbot in Python, it's essential to have a solid understanding of natural language processing (NLP) concepts and techniques. NLP involves processing and understanding human language, allowing the chatbot to interact with users effectively. Key concepts include tokenization, part-of-speech tagging, named entity recognition, and sentiment analysis. Python provides a range of libraries, such as NLTK (Natural Language Toolkit) and spaCy, which can be used to implement these NLP functionalities.

2. Choosing a Framework: There are various frameworks available in Python that simplify the process of building a chatbot. One popular option is the ChatterBot library, which provides a straightforward way to implement conversational agents. ChatterBot allows you to train a chatbot using a large dataset of conversations and provides functionalities for generating appropriate responses. Other options like Rasa and Dialogflow also offer rich features for building chatbots with advanced capabilities like natural language understanding, intent recognition, and context management.

3. Implementing the Chatbot: Once you have chosen the framework, it's time to implement the chatbot. This involves defining the chatbot's responses based on the user's input, handling various conversational scenarios, and integrating the chatbot into your desired platform or interface. You can train the chatbot using existing datasets or create your own dataset.

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

Given Data Set:

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

Here's a list of tools and software commonly used in theprocess:

1. **Python:**

   The programming language itself is essential for building the chatbot.

2. **Python Libraries:**

- **NLTK (Natural Language Toolkit):** For natural language processing and text analysis.
- **spaCy:** Another NLP library for various NLP tasks.
- **Gensim:** For topic modeling and document similarity analysis.
- **Scikit-learn:** For machine learning and data preprocessing.
- **TensorFlow or PyTorch:** For creating and training deep learning models for chatbot responses.
- **Keras:** High-level API that works on top of TensorFlow and makes it easier to build neural networks.

3. **Chatbot Frameworks:**

- **ChatterBot:** A Python library for creating chatbots using machine learning.
- **Rasa:** An open-source chatbot framework for creating conversational AI with natural language understanding and dialogue management.
- **Dialogflow (formerly API.ai):** A cloud-based chatbot development platform by Google.

4. **Text Editors/IDEs:**
   - PyCharm, Visual Studio Code, Jupyter Notebook, or any other code editor or integrated development environment.

5. **Version Control:**
   Git and GitHub/GitLab/Bitbucket: For version control and collaboration with other developers.

6. **Web Frameworks (if building web-based chatbots):**
   - Django, Flask, or FastAPI: For creating web applications to interact with the chatbot.

7. Database (for storing chatbot data):

SQLite, MySQL, PostgreSQL, or other relational databases.

NoSQL databases like MongoDB for unstructured data.

8. APIs (for integrating external services):

REST APIs, GraphQL, and other APIs relevant to your chatbot's purpose.

9. Front-End Tools (if building a web-based chatbot):

HTML, CSS, and JavaScript: For creating the user interface and integrating with the chatbot.

10. Server/Hosting:

Heroku, AWS, Google Cloud, or other cloud platforms for deploying and hosting your chatbot.

Continuous Integration/Continuous Deployment (CI/CD):

Tools like Jenkins, Travis CI, or CircleCI to automate the deployment process.

Natural Language Understanding Services (if required):

Google Cloud Natural Language API, Microsoft Azure Text Analytics, IBM Watson NLU, or other cloud-based NLU services.

11. Testing and Debugging Tools:

PyTest or other testing frameworks.

Logging and Debugging tools to monitor and troubleshoot the chatbot's performance.

12. Version Control Tools:

Git with platforms like GitHub, GitLab, or Bitbucket for version control and collaboration.

13. Data Annotation Tools:

Tools like Prodigy, Labelbox, or Doccano for annotating training data.

14.Speech Recognition and Text-to-Speech Tools (if needed):

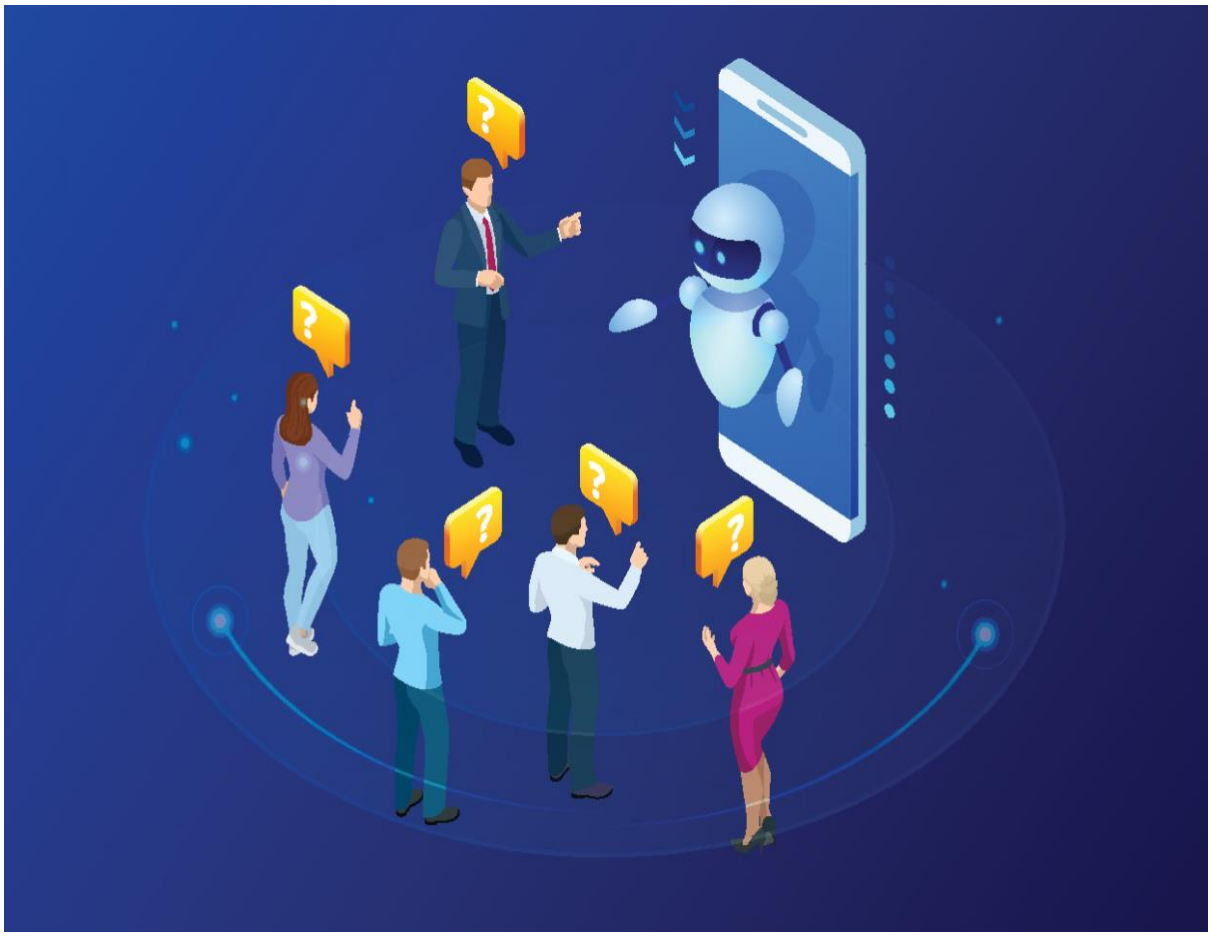
Google Cloud Speech-to-Text and Text-to-Speech, IBM Watson Speech to Text, and Amazon Polly.

15.Conversational Design Tools:

Botmock, Botsociety, or Adobe XD for designing chatbot conversations.

16.Knowledge Bases and APIs:

Access to external data sources and knowledge bases for providing accurate responses.



1. Design Thinking and Present in Form of Document

1. Introduction:

Design thinking is a problem-solving approach that is focused on understanding and empathizing with users, defining their needs, and creating innovative solutions. In this document, we will apply the principles of design thinking to create a Chatbot using Python.

2. Empathize:

To begin, we need to empathize with the target users of our Chatbot. Understand their pain points, needs, and expectations. Conduct user research, interviews, and surveys to gather insights. Identify the specific problem that the Chatbot will address.

3. Define:

Based on the insights gathered, define a clear problem statement for your Chatbot. For example, "The Chatbot will provide personalized recommendations for restaurants based on user preferences and location."

4. Ideate:

Generate multiple ideas and potential solutions for your Chatbot. Encourage brainstorming sessions and involve a diverse team to explore different possibilities. Consider functionality, user interface, and integration with other platforms or systems.

5. Prototype:

Create a low-fidelity prototype of your Chatbot using wireframing or mockup tools. Focus on the core features and functionalities. Test the prototype with a small group of users to gather feedback and iterate.

6. Test:

Conduct usability testing sessions with representative users to evaluate the effectiveness of your Chatbot. Pay attention to user experience, ease of use, and the ability to meet the defined problem statement. Gather feedback and make necessary improvements.

2.Design into Innovation

Introduction:

Chatbots have become increasingly popular in recent years, revolutionizing the way businesses and individuals interact with applications and services. In this guide, we will explore the design process and implementation steps for creating a chatbot using Python, a widely-used and versatile programming language.

1. Define the Purpose:

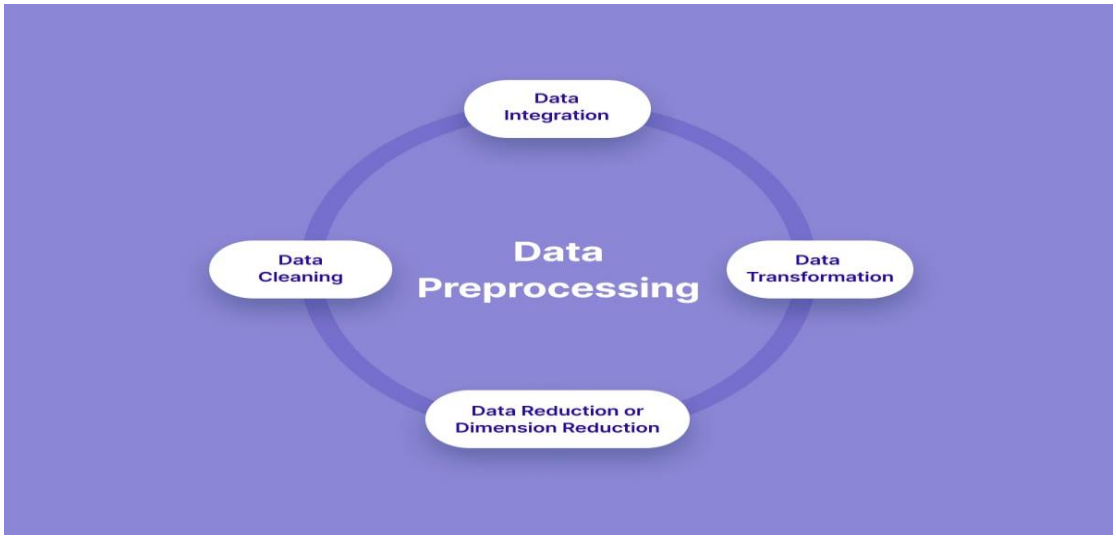
Before diving into the design and development process, it is crucial to define the purpose and goals of your chatbot. Consider the following questions:

- What is the primary objective of your chatbot? Is it to provide customer support, answer frequently asked questions, or assist in making reservations or purchases?

- Who is the target audience for your chatbot? Understanding the demographics and specific needs of your users will help tailor the chatbot's functionality and conversational style.

2. Design the Conversational Flow:

Once you have a clear purpose in mind, it's time to design the conversational flow of your chatbot. Start by creating a list of possible user inputs and corresponding chatbot responses. Consider different scenarios and potential user intents. Use tools like flowcharts or diagrams to visualize the flow of the conversation.



Python program:

```
from tensorflow.keras.layers import TextVectorization
import re,string
from tensorflow.keras.layers import LSTM,Dense,Embedding,Dropout,LayerNormaliz
ation
#data preprocessing
df=pd.read_csv('/kaggle/input/simple-dialogs-for-chatbot/dialogs.txt',sep='\t'
,names=['question','answer'])
print(f'Dataframe size: {len(df)}')
df.head()
df['question tokens']=df['question'].apply(lambda x:len(x.split()))
df['answer tokens']=df['answer'].apply(lambda x:len(x.split()))
plt.style.use('fivethirtyeight')
fig,ax=plt.subplots(nrows=1,ncols=2,figsize=(20,5))
sns.set_palette('Set2')
sns.histplot(x=df['question tokens'],data=df,kde=True,ax=ax[0])
sns.histplot(x=df['answer tokens'],data=df,kde=True,ax=ax[1])
sns.jointplot(x='question tokens',y='answer tokens',data=df,kind='kde',fill=Tr
ue,cmap='YlGnBu')
plt.show()
def clean_text(text):
    text=re.sub('-',' ',text.lower(
    text=re.sub('[.]',' . ',text)
    text=re.sub('[1]',' 1 ',text)
    text=re.sub('[2]',' 2 ',text)
    text=re.sub('[3]',' 3 ',text)
    text=re.sub('[4]',' 4 ',text)
    text=re.sub('[5]',' 5 ',text)
    return text
df.drop(columns=['answer tokens','question tokens'],axis=1,inplace=True)
df['encoder_inputs']=df['question'].apply(clean_text)
df['decoder_targets']=df['answer'].apply(clean_text)+' <end>'
df['decoder_inputs']='<start> '+df['answer'].apply(clean_text)+' <end>'
df.head(10)
df['encoder input tokens']=df['encoder_inputs'].apply(lambda x:len(x.split()))
df['decoder input tokens']=df['decoder_inputs'].apply(lambda x:len(x.split()))
df['decoder target tokens']=df['decoder_targets'].apply(lambda x:len(x.split()
))
plt.style.use('fivethirtyeight')
fig,ax=plt.subplots(nrows=1,ncols=3,figsize=(20,5))
sns.set_palette('Set2')
sns.histplot(x=df['encoder input tokens'],data=df,kde=True,ax=ax[0])
sns.histplot(x=df['decoder input tokens'],data=df,kde=True,ax=ax[1])
sns.histplot(x=df['decoder target tokens'],data=df,kde=True,ax=ax[2])
sns.jointplot(x='encoder input tokens',y='decoder target tokens',data=df,kind=
'kde',fill=True,cmap='YlGnBu')
plt.show()
print(f"After preprocessing: {' '.join(df[df['encoder input tokens'].max()==df
['encoder input tokens']]['encoder_inputs'].values.tolist())}")
print(f"Max encoder input length: {df['encoder input tokens'].max()}")
print(f"Max decoder input length: {df['decoder input tokens'].max()}")
print(f"Max decoder target length: {df['decoder target tokens'].max()}")
df.drop(columns=['question','answer','encoder input tokens','decoder input tok
ens','decoder target tokens'],axis=1,inplace=True)
params={
```

```
    "vocab_size":2500,
    "max_sequence_length":30,
    "learning_rate":0.008,
    "batch_size":149,}
learning_rate=params['learning_rate']
batch_size=params['batch_size']
embedding_dim=params['embedding_dim']
lstm_cells=params['lstm_cells']
vocab_size=params['vocab_size']
buffer_size=params['buffer_size']
max_sequence_length=params['max_sequence_length']
df.head(10)
vectorize_layer=TextVectorization(
    max_tokens=vocab_size,
    standardize=None,
    output_mode='int',
    output_sequence_length=max_sequence_length
)
vectorize_layer.adapt(df['encoder_inputs']+' '+df['decoder_targets']+' <start>
<end>')
vocab_size=len(vectorize_layer.get_vocabulary())
print(f'Vocab size: {len(vectorize_layer.get_vocabulary())}')
print(f'{vectorize_layer.get_vocabulary()[:12]}')
def sequences2ids(sequence):
    return vectorize_layer(sequence)
def ids2sequences(ids):
    decode=''
    if type(ids)==int:
        ids=[ids]
    for id in ids:
        decode+=vectorize_layer.get_vocabulary()[id]+' '
    return decode

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

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.
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.

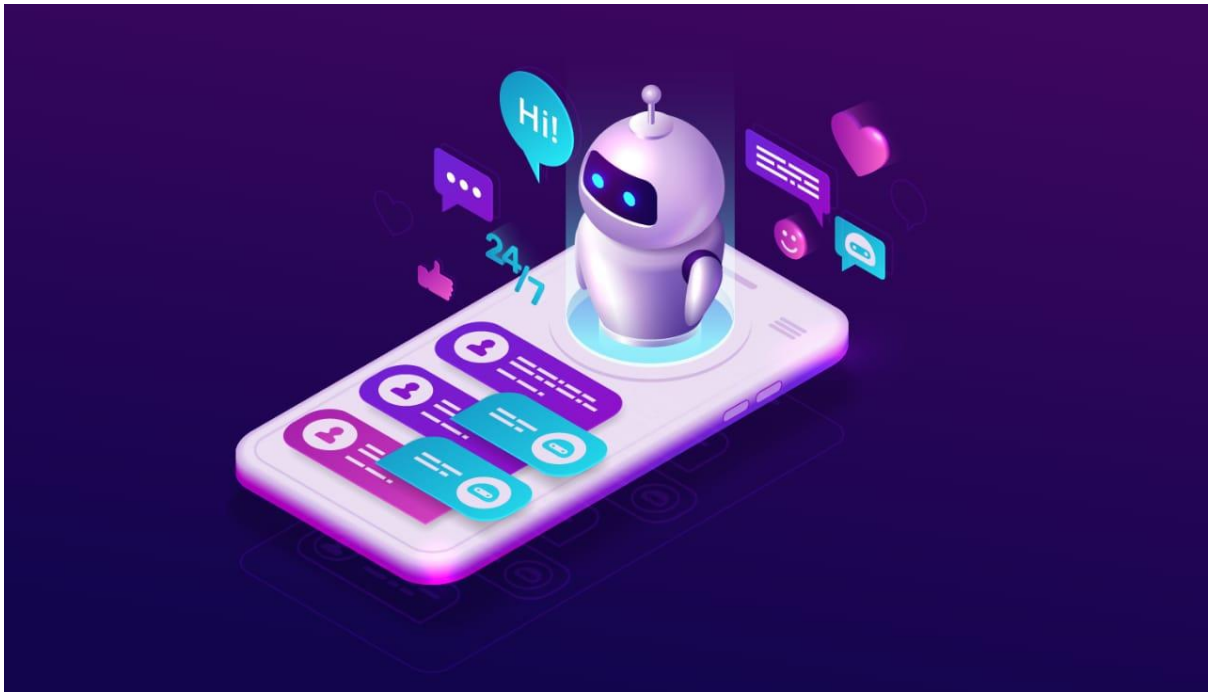
i go to pcc. do you like it there?
do you like it there? it's okay. it's a really big campus.

After preprocessing:

for example , if your birth date is january 1 2 , 1 9 8 7 , write 0 1 / 1 2 / 8 7 .
Max encoder input length: 27
Max decoder input length: 29
Max decoder target length: 28
Question sentence: hi , how are you ?
Question to tokens: [1971 9 45 24 8 7 0 0 0 0]
Encoder input shape: (3725, 30)
Decoder input shape: (3725, 30)
Decoder target shape: (3725, 30)

3. Choose a Framework or Library:

Python offers several frameworks and libraries for developing chatbots. One popular choice is the Natural Language Toolkit (NLTK), which provides a suite of libraries and programs for natural language processing. Another option is the ChatterBot library, which simplifies the process of creating chatbots using pre-trained machine learning models.



3.Build loading and preprocessing the dataset

In the context of creating a chatbot in Python, "build loading" refers to the process of importing or loading the necessary libraries, modules, or packages required for building a chatbot. This typically includes importing libraries such as NLTK (Natural Language Toolkit) for natural language processing or TensorFlow for machine learning.

On the other hand, "preprocessing the data set" refers to the steps taken to clean, transform, and prepare the data before it can be used for training a chatbot. This involves tasks such as removing unnecessary characters or symbols, tokenizing the text into individual words, removing stop words, and performing other text normalization techniques like lemmatization or stemming.

By combining build loading and preprocessing steps, you ensure that you have the necessary tools and resources to build the chatbot, and that the training data is properly cleaned and formatted for further processing.

Python program:

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

    def call(self,encoder_inputs):
        self.inputs=encoder_inputs
        x=self.embedding(encoder_inputs)
        x=self.normalize(x)
```

```
        x=Dropout(.4)(x)
        encoder_outputs,encoder_state_h,encoder_state_c=self.lstm(x)
        self.outputs=[encoder_state_h,encoder_state_c]
        return encoder_state_h,encoder_state_c

encoder=Encoder(lstm_cells,embedding_dim,vocab_size,name='encoder')
encoder.call(_[0])
class Decoder(tf.keras.models.Model):
    def __init__(self,units,embedding_dim,vocab_size,*args,**kwargs) -> None:
        super().__init__(*args,**kwargs)
        self.units=units
        self.embedding_dim=embedding_dim
        self.vocab_size=vocab_size
        self.embedding=Embedding(
            vocab_size,
            embedding_dim,
            name='decoder_embedding',
            mask_zero=True,
            embeddings_initializer=tf.keras.initializers.HeNormal()
        )
        self.normalize=LayerNormalization()
        self.lstm=LSTM(
            units,
            dropout=.4,
            return_state=True,
            return_sequences=True,
            name='decoder_lstm',
            kernel_initializer=tf.keras.initializers.HeNormal()
        )
        self.fc=Dense(
            vocab_size,
            activation='softmax',
            name='decoder_dense',
            kernel_initializer=tf.keras.initializers.HeNormal()
        )

    def call(self,decoder_inputs,encoder_states):
        x=self.embedding(decoder_inputs)
        )
decoder=Decoder(lstm_cells,embedding_dim,vocab_size,name='decoder')
decoder(_[1][:1],encoder(_[0][:1]))
class ChatBotTrainer(tf.keras.models.Model):
    def __init__(self,encoder,decoder,*args,**kwargs):
        super().__init__(*args,**kwargs)
        self.encoder=encoder
        self.decoder=decoder

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

    def accuracy_fn(self,y_true,y_pred):
        pred_values = tf.cast(tf.argmax(y_pred, axis=-1), dtype='int64')
        correct = tf.cast(tf.equal(y_true, pred_values), dtype='float64')
```

```
        mask = tf.cast(tf.greater(y_true, 0), dtype='float64')
    def call(self,inputs):
        encoder_inputs,decoder_inputs=inputs
        encoder_states=self.encoder(encoder_inputs)
        return self.decoder(decoder_inputs,encoder_states)
```

```
model=ChatBotTrainer(encoder,decoder,name='chatbot_trainer')
model.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
    weighted_metrics=['loss','accuracy']
)
    callbacks=[
        tf.keras.callbacks.TensorBoard(log_dir='logs'),
        tf.keras.callbacks.ModelCheckpoint('ckpt',verbose=1,save_best_only=Tru
e)
    ]
)
```

OUTPUT:

```
(<tf.Tensor: shape=(149, 256), dtype=float32, numpy=
 array([[ 0.16966951, -0.10419625, -0.12700348, ..., -0.12251794,
         0.10568858,  0.14841646],
       [ 0.08443093,  0.08849293, -0.09065959, ..., -0.00959182,
         0.10152507, -0.12077457],
       [ 0.03628462, -0.02653611, -0.11506603, ..., -0.14669597,
         0.10292757,  0.13625325],
<tf.Tensor: shape=(1, 30, 2443), dtype=float32, numpy=
array([[[3.4059247e-04, 5.7348556e-05, 2.1294907e-05, ...,
         7.2067953e-05, 1.5453645e-03, 2.3599296e-04],
        [1.4662130e-03, 8.0250365e-06, 5.4062020e-05, ...,
         1.9187471e-05, 9.7244098e-05, 7.6433855e-05],
        [9.6929165e-05, 2.7441782e-05, 1.3761305e-03,
tf.Tensor: shape=(149, 30, 2443), dtype=float32, numpy=
array([[[3.40592262e-04, 5.73484940e-05, 2.12948853e-05, ...,
         7.20679745e-05, 1.54536311e-03, 2.35993255e-04],
        [1.46621116e-03, 8.02504110e-06, 5.40619949e-05, ...,
         1.91874733e-05, 9.72440175e-05, 7.64339056e-05],
        [9.69291723e-05, 2.74417835e-05, 1.37613132e-03, ...,
         3.60095728e-05, 1.55378671e-04, 1.83973272e-04],
        ...,
Epoch 1/100
23/23 [==============================] - ETA: 0s - loss: 1.6590 - accuracy: 0.2180
Epoch 1: val_loss improved from inf to 1.21875, saving model to ckpt
23/23 [==============================] - 68s 3s/step - loss: 1.6515 - accuracy: 0.21
98 - val_loss: 1.2187 - val_accuracy: 0.3072
Epoch 2/100
23/23 [==============================] - ETA: 0s - loss: 1.2327 - accuracy: 0.3087
Epoch 2: val_loss improved from 1.21875 to 1.10877, saving model to ckpt
```

4.Performing different activities like feature engineering, model training,evaluation etc..,

FEATURE ENGINEERING:

Feature engineering refers to the process of extracting meaningful features from the input text data to enhance the performance and accuracy of the chatbot.

1. Tokenization: Split the text into individual words or tokens. This is important for further analysis and processing.

2. Stopword Removal: Remove common words that do not carry much meaning, such as articles, prepositions, and conjunctions. This helps to focus on the more informative content of the input.

3. Lemmatization/Stemming: Reduce words to their base or root form. This helps to handle different variations of the same word and improves the chatbot's ability to understand user input.

4. Part-of-Speech (POS) Tagging: Assign grammatical tags to each word in the text, such as noun, verb, adjective, etc. This can help identify the structure and context of the user's input.

5. Named Entity Recognition (NER): Identify and classify named entities in the text, such as names, locations, organizations, etc. This can be useful for providing specific responses or performing targeted actions based on recognized entities.

6. Sentiment Analysis: Determine the sentiment or emotion expressed in the user's input. This can help the chatbot tailor its responses accordingly.

7. Word2Vec/Doc2Vec Embeddings: Convert words or documents into dense numerical vectors that capture semantic meaning. This can help the chatbot understand .

Data preprocessing and visualisation:

Data preprocessing is the process of transforming and manipulating raw data to make it suitable for analysis, while data visualization refers to the graphical representation of data to gain insights and communicate information effectively in the context of creating a chatbot in Python.

Python program:

#DATA PREPROCESSING:

In [1]:

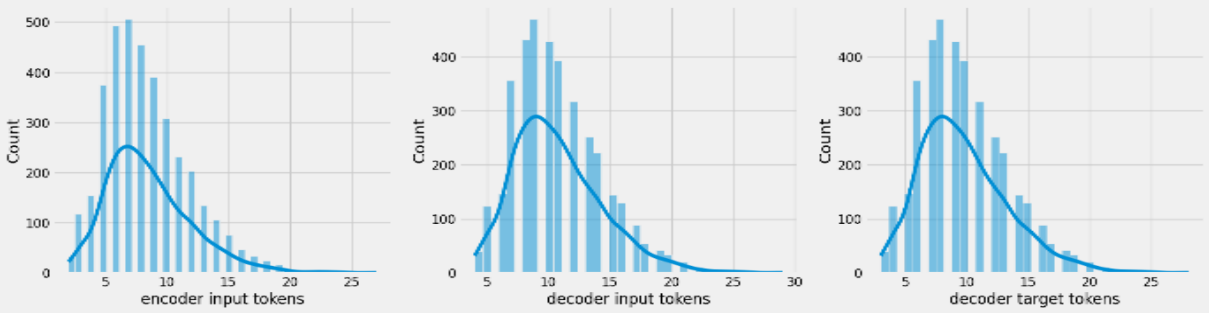
```
from tensorflow.keras.layers import TextVectorization
import re,string
from tensorflow.keras.layers import LSTM,Dense,Embedding,Dropout,LayerNormaliz
ation
#data preprocessing
df=pd.read_csv('/kaggle/input/simple-dialogs-for-chatbot/dialogs.txt',sep='\t'
,names=['question','answer'])
print(f'Dataframe size: {len(df)}')
df.head()
df['question tokens']=df['question'].apply(lambda x:len(x.split()))
df['answer tokens']=df['answer'].apply(lambda x:len(x.split()))
plt.style.use('fivethirtyeight')
fig,ax=plt.subplots(nrows=1,ncols=2,figsize=(20,5))
sns.set_palette('Set2')
sns.histplot(x=df['question tokens'],data=df,kde=True,ax=ax[0])
sns.histplot(x=df['answer tokens'],data=df,kde=True,ax=ax[1])
sns.jointplot(x='question tokens',y='answer tokens',data=df,kind='kde',fill=Tr
ue,cmap='YlGnBu')
plt.show()
def clean_text(text):
    text=re.sub('-',' ',text.lower(
    text=re.sub('[.]',' . ',text)
    text=re.sub('[1]',' 1 ',text)
    text=re.sub('[2]',' 2 ',text)
    text=re.sub('[3]',' 3 ',text)
    text=re.sub('[4]',' 4 ',text)
    text=re.sub('[5]',' 5 ',text)
    return text
df.drop(columns=['answer tokens','question tokens'],axis=1,inplace=True)
df['encoder_inputs']=df['question'].apply(clean_text)
df['decoder_targets']=df['answer'].apply(clean_text)+' <end>'
df['decoder_inputs']='<start> '+df['answer'].apply(clean_text)+' <end>'
df.head(10)
df['encoder input tokens']=df['encoder_inputs'].apply(lambda x:len(x.split()))
df['decoder input tokens']=df['decoder_inputs'].apply(lambda x:len(x.split()))
df['decoder target tokens']=df['decoder_targets'].apply(lambda x:len(x.split()
))
plt.style.use('fivethirtyeight')
fig,ax=plt.subplots(nrows=1,ncols=3,figsize=(20,5))
sns.set_palette('Set2')
sns.histplot(x=df['encoder input tokens'],data=df,kde=True,ax=ax[0])
sns.histplot(x=df['decoder input tokens'],data=df,kde=True,ax=ax[1])
sns.histplot(x=df['decoder target tokens'],data=df,kde=True,ax=ax[2])
sns.jointplot(x='encoder input tokens',y='decoder target tokens',data=df,kind=
'kde',fill=True,cmap='YlGnBu')
plt.show()
print(f"After preprocessing: {' '.join(df[df['encoder input tokens'].max()==df
['encoder input tokens']]['encoder_inputs'].values.tolist())}")
print(f"Max encoder input length: {df['encoder input tokens'].max()}")
```

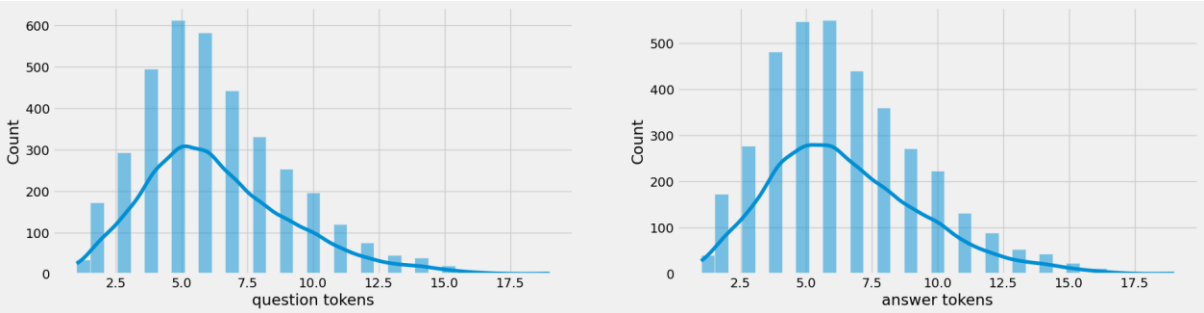
```
print(f"Max decoder input length: {df['decoder input tokens'].max()}")
print(f"Max decoder target length: {df['decoder target tokens'].max()}")
df.drop(columns=['question','answer','encoder input tokens','decoder input tok
ens','decoder target tokens'],axis=1,inplace=True)
params={
    "vocab_size":2500,
    "max_sequence_length":30,
    "learning_rate":0.008,
    "batch_size":149,}
learning_rate=params['learning_rate']
batch_size=params['batch_size']
embedding_dim=params['embedding_dim']
lstm_cells=params['lstm_cells']
vocab_size=params['vocab_size']
buffer_size=params['buffer_size']
max_sequence_length=params['max_sequence_length']
df.head(10)
vectorize_layer=TextVectorization(
    max_tokens=vocab_size,
    standardize=None,
    output_mode='int',
    output_sequence_length=max_sequence_length
)
vectorize_layer.adapt(df['encoder_inputs']+' '+df['decoder_targets']+' <start>
<end>')
vocab_size=len(vectorize_layer.get_vocabulary())
print(f'Vocab size: {len(vectorize_layer.get_vocabulary())}')
print(f'{vectorize_layer.get_vocabulary()[:12]}')
def sequences2ids(sequence):
    return vectorize_layer(sequence)
def ids2sequences(ids):
    decode=''
    if type(ids)==int:
        ids=[ids]
    for id in ids:
        decode+=vectorize_layer.get_vocabulary()[id]+' '
    return decode

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

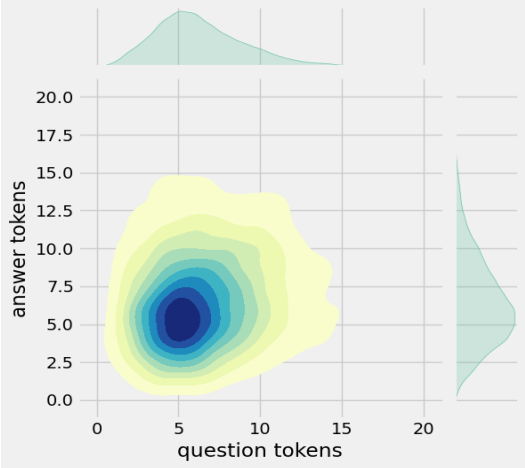
Out [1]:

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.







After preprocessing: for example , if your birth date is january 1 2 , 1 9 8 7 , write 0 1 / 1 2 / 8 7 .
Max encoder input length: 27
Max decoder input length: 29
Max decoder target length: 28
Question sentence: hi , how are you ?
Question to tokens: [1971 9 45 24 8 7 0 0 0 0]
Encoder input shape: (3725, 30)

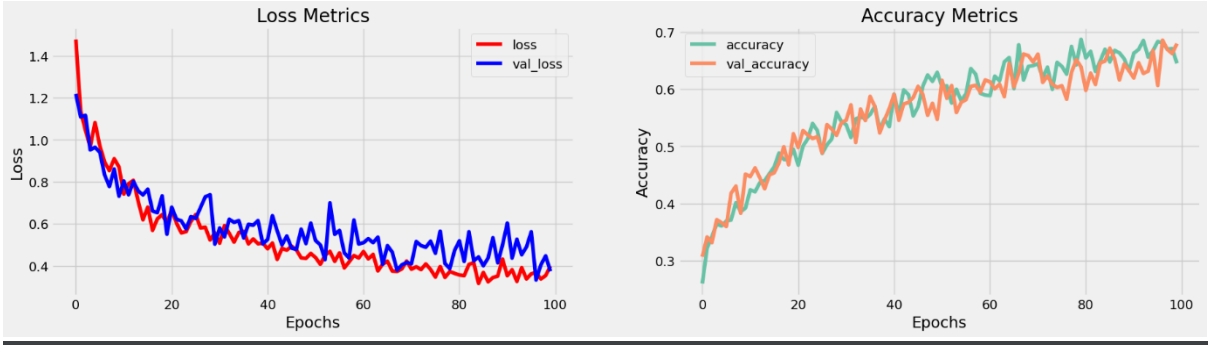
Decoder input shape: (3725, 30)
Decoder target shape: (3725, 30)

#Visualizing metrics

In [2]:

```
fig,ax=plt.subplots(nrows=1,ncols=2,figsize=(20,5))
ax[0].plot(history.history['loss'],label='loss',c='red')
ax[0].plot(history.history['val_loss'],label='val_loss',c = 'blue')
ax[0].set_xlabel('Epochs')
ax[1].set_xlabel('Epochs')
ax[0].set_ylabel('Loss')
ax[1].set_ylabel('Accuracy')
ax[0].set_title('Loss Metrics')
ax[1].set_title('Accuracy Metrics')
ax[1].plot(history.history['accuracy'],label='accuracy')
ax[1].plot(history.history['val_accuracy'],label='val_accuracy')
ax[0].legend()
ax[1].legend()
plt.show()
```

Out [2]:



Model selection

There are several popular models and frameworks available for developing chatbots in Python. Here are a few options you can consider:

1. Natural Language Toolkit (NLTK): NLTK is a widely used library in Python for natural language processing (NLP) tasks. It provides various tools and algorithms for text classification, tokenization, stemming, and more. NLTK can be a good choice if you want to build a simple rule-based chatbot.

2. ChatterBot: ChatterBot is a Python library that uses machine learning algorithms to generate responses based on training data. It supports multiple languages, and you can train the chatbot using your own dataset or use pre-trained models. ChatterBot can be a good option if you want to create a chatbot with a learning capability.

3. TensorFlow and Keras: These are popular libraries for building neural networks and deep learning models. You can use them to create a chatbot with a more advanced natural language understanding and generation capability. TensorFlow and Keras provide various pre-trained models and techniques that can be utilized for chatbot development.

4. Rasa: Rasa is an open-source framework that offers both natural language understanding (NLU) and dialogue management capabilities for building chatbots. It allows you to design conversational flows, handle user intents, and store contextual information. Rasa is suitable for building more complex and interactive.

MODEL TRAINING:

1. Collect and preprocess your training data: Find a dataset that contains creative and diverse conversations. This could include chat logs, social media conversations, or any other relevant source. Make sure to clean the data by removing any noise or irrelevant information.

2. Install the necessary libraries: Python offers several libraries for implementing chatbots. Some popular ones include NLTK (Natural Language Toolkit), spaCy, and TensorFlow. Install the required libraries using pip or conda.

3. Define your chatbot architecture: Decide on the architecture you want your chatbot to have. This could be a rule-based system, retrieval-based model, or even a generative model using techniques like sequence-to-sequence with attention.

4. Preprocess your training data: Convert your text data into a suitable format for training. This involves tokenizing the text, splitting it into sentences, and performing any other necessary preprocessing steps like removing stop words or performing lemmatization.

5. Build and train your chatbot model: Implement your chosen chatbot architecture using Python and the libraries you installed. Train your model on the preprocessed training data. This typically involves feeding the input text to the model and optimizing its parameters using techniques like gradient descent.

6. Evaluate and fine-tune your model: After training, evaluate your chatbot's performance using appropriate metrics like perplexity or BLEU score.

MODEL EVALUATION:

Model evaluation for a chatbot created in Python, Model evaluation is crucial to assess the pe rformance and effectiveness of your chatbot. Here are the steps you can follow:

1. Define evaluation metrics: Determine the key metrics you want to use to evaluate your ch atbot. Common metrics include precision, recall, F1 score, accuracy, and perplexity. The choi ce of metrics depends on the specific goals of your chatbot.

2. Prepare test data: Collect a set of test data that represents the kind of conversations your chatbot is expected to handle. This data should cover a wide range of scenarios and be repres entative of potential user inputs and expected responses.

3. Split the data: Divide your test data into two parts: a development set and a test set. The d evelopment set will be used for fine-tuning and optimizing your chatbot, while the test set wil l be used for the final evaluation.

4. Implement the evaluation code: Write Python code to evaluate the performance of your c hatbot. This code should load your trained chatbot model, process the test data, and calculate the evaluation metrics you defined earlier.

5. Evaluate the chatbot: Run the evaluation code on your test data and analyze the results. Measure the performance of your chatbot using the defined metrics. You may also conduct a qualitative analysis by manually reviewing some example conversations.

6. Iterate and improve: Based on the evaluation results, make necessary improvements .

In [3]:

```
model.load_weights('ckpt')
model.save('models',save_format='tf')
```

```
linkcode
for idx,i in enumerate(model.layers):
    print('Encoder layers:' if idx==0 else 'Decoder layers: ')
    for j in i.layers:
        print(j)
    print('---------------------')
```

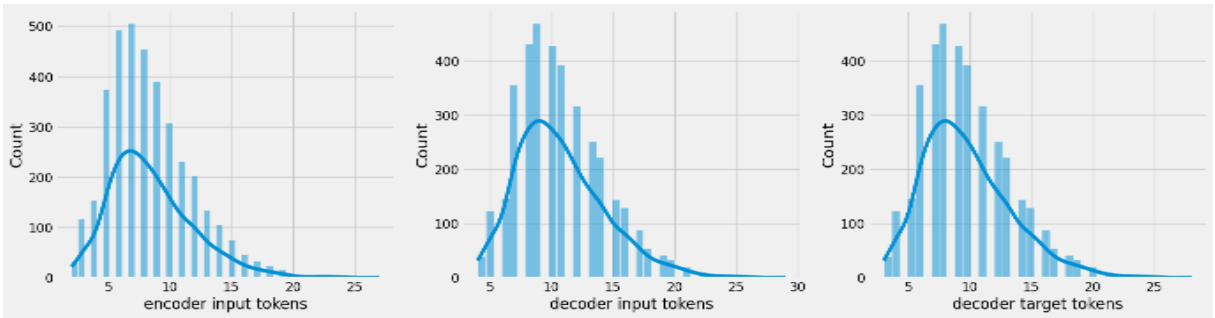
In [4]:

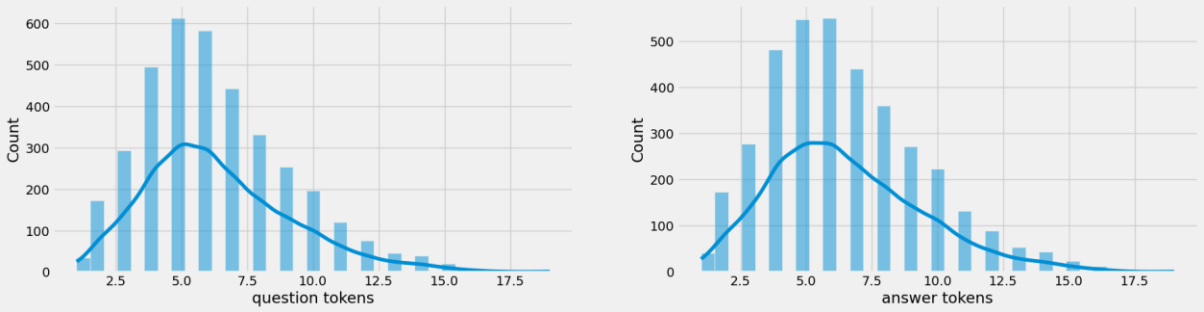
```
from tensorflow.keras.layers import TextVectorization
import re,string
```

```
from tensorflow.keras.layers import LSTM,Dense,Embedding,Dropout,LayerNormaliz
ation
#data preprocessing
df=pd.read_csv('/kaggle/input/simple-dialogs-for-chatbot/dialogs.txt',sep='\t'
,names=['question','answer'])
print(f'Dataframe size: {len(df)}')
df.head()
df['question tokens']=df['question'].apply(lambda x:len(x.split()))
df['answer tokens']=df['answer'].apply(lambda x:len(x.split()))
plt.style.use('fivethirtyeight')
fig,ax=plt.subplots(nrows=1,ncols=2,figsize=(20,5))
sns.set_palette('Set2')
sns.histplot(x=df['question tokens'],data=df,kde=True,ax=ax[0])
sns.histplot(x=df['answer tokens'],data=df,kde=True,ax=ax[1])
sns.jointplot(x='question tokens',y='answer tokens',data=df,kind='kde',fill=Tr
ue,cmap='YlGnBu')
plt.show()
def clean_text(text):
    text=re.sub('-',' ',text.lower(
    text=re.sub('[.]',' . ',text)
    text=re.sub('[1]',' 1 ',text)
    text=re.sub('[2]',' 2 ',text)
    text=re.sub('[3]',' 3 ',text)
    text=re.sub('[4]',' 4 ',text)
    text=re.sub('[5]',' 5 ',text)
    return text
df.drop(columns=['answer tokens','question tokens'],axis=1,inplace=True)
df['encoder_inputs']=df['question'].apply(clean_text)
df['decoder_targets']=df['answer'].apply(clean_text)+' <end>'
df['decoder_inputs']='<start> '+df['answer'].apply(clean_text)+' <end>'
df.head(10)
df['encoder input tokens']=df['encoder_inputs'].apply(lambda x:len(x.split()))
df['decoder input tokens']=df['decoder_inputs'].apply(lambda x:len(x.split()))
df['decoder target tokens']=df['decoder_targets'].apply(lambda x:len(x.split()
))
plt.style.use('fivethirtyeight')
fig,ax=plt.subplots(nrows=1,ncols=3,figsize=(20,5))
sns.set_palette('Set2')
sns.histplot(x=df['encoder input tokens'],data=df,kde=True,ax=ax[0])
sns.histplot(x=df['decoder input tokens'],data=df,kde=True,ax=ax[1])
sns.histplot(x=df['decoder target tokens'],data=df,kde=True,ax=ax[2])
sns.jointplot(x='encoder input tokens',y='decoder target tokens',data=df,kind=
'kde',fill=True,cmap='YlGnBu')
plt.show()
print(f"After preprocessing: {' '.join(df[df['encoder input tokens'].max()==df
['encoder input tokens']]['encoder_inputs'].values.tolist())}")
print(f"Max encoder input length: {df['encoder input tokens'].max()}")
print(f"Max decoder input length: {df['decoder input tokens'].max()}")
print(f"Max decoder target length: {df['decoder target tokens'].max()}")
df.drop(columns=['question','answer','encoder input tokens','decoder input tok
ens','decoder target tokens'],axis=1,inplace=True)
params={
    "vocab_size":2500,
    "max_sequence_length":30,
    "learning_rate":0.008,
    "batch_size":149,}
```

```
learning_rate=params['learning_rate']
batch_size=params['batch_size']
embedding_dim=params['embedding_dim']
lstm_cells=params['lstm_cells']
vocab_size=params['vocab_size']
buffer_size=params['buffer_size']
max_sequence_length=params['max_sequence_length']
df.head(10)
vectorize_layer=TextVectorization(
    max_tokens=vocab_size,
    standardize=None,
    output_mode='int',
    output_sequence_length=max_sequence_length
)
vectorize_layer.adapt(df['encoder_inputs']+' '+df['decoder_targets']+' <start>
<end>')
vocab_size=len(vectorize_layer.get_vocabulary())
print(f'Vocab size: {len(vectorize_layer.get_vocabulary())}')
print(f'{vectorize_layer.get_vocabulary()[:12]}')
def sequences2ids(sequence):
    return vectorize_layer(sequence)
def ids2sequences(ids):
    decode=''
    if type(ids)==int:
        ids=[ids]
    for id in ids:
        decode+=vectorize_layer.get_vocabulary()[id]+' '
    return decode

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

Out [4]:





#SAVE MODEL

In [5]:

```
model.load_weights('ckpt')
model.save('models',save_format='tf')
```

```
In [18]:
linkcode
for idx,i in enumerate(model.layers):
    print('Encoder layers:' if idx==0 else 'Decoder layers: ')
    for j in i.layers:
        print(j)
    print('---------------------')
```

Out [5]:

```
Encoder layers:
<keras.layers.core.embedding.Embedding object at 0x782084b9d190>
>
---------------------
Decoder layers:
<keras.layers.core.embedding.Embedding object at 0x78207c258590>
<keras.layers.normalization.layer_normalization.LayerNormalization object at 0
x78207c78bd10>
<keras.layers.rnn.lstm.LSTM object at 0x78207c258a10>
------------------
```

#CREATE INFERENCE MODEL

In [6]:

```
class ChatBot(tf.keras.models.Model):
    def __init__(self,base_encoder,base_decoder,*args,**kwargs):
        super().__init__(*args,**kwargs)
        self.encoder,self.decoder=self.build_inference_model(base_encoder,base
_decoder)

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

        decoder_input_state_h=tf.keras.Input(shape=(lstm_cells,))
```

```
        decoder_input_state_c=tf.keras.Input(shape=(lstm_cells,))
        decoder_inputs=tf.keras.Input(shape=(None,))
        x=base_decoder.layers[0](decoder_inputs)
        x=base_encoder.layers[1](x)
        x,decoder_state_h,decoder_state_c=base_decoder.layers[2](x,initial_sta
te=[decoder_input_state_h,decoder_input_state_c])
        decoder_outputs=base_decoder.layers[-1](x)
        decoder=tf.keras.models.Model(
            inputs=[decoder_inputs,[decoder_input_state_h,decoder_input_state_
c]],
            outputs=[decoder_outputs,[decoder_state_h,decoder_state_c]],name='
chatbot_decoder'
        )
        return encoder,decoder

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

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

    def sample(self,conditional_probability,temperature=0.5):
        conditional_probability = np.asarray(conditional_probability).astype("
float64")
       conditional_probability = np.log(conditional_probability) / temperature
chatbot.summary()
tf.keras.utils.plot_model(chatbot.encoder,to_file='encoder.png',show_shapes=Tr
ue,show_layer_activations=True)
tf.keras.utils.plot_model(chatbot.decoder,to_file='decoder.png',show_shapes=True,s
how_layer_activations=True)
```

Out [6]:

Model: "chatbot_encoder"

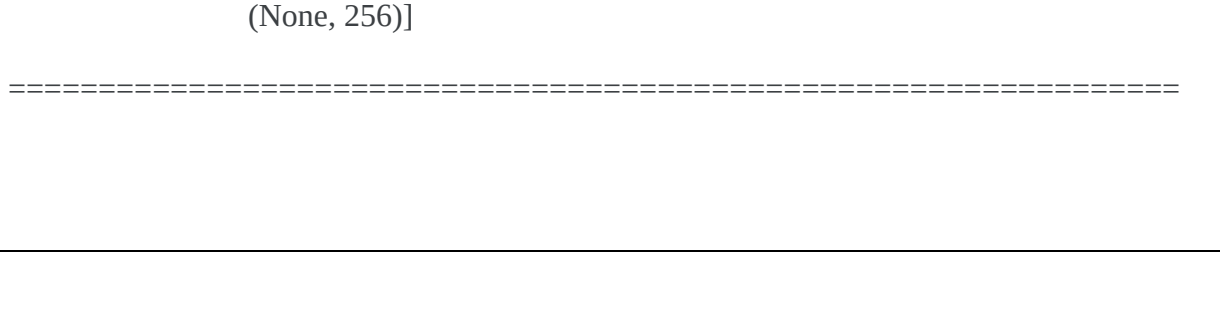
Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, None)]	0
encoder_embedding (Embeddin g)	(None, None, 256)	625408
layer_normalization (LayerN ormalization)	(None, None, 256)	512
encoder_lstm (LSTM)	[(None, None, 256), (None, 256), (None, 256)]	525312

Total params: 1,151,232
Trainable params: 1,151,232
Non-trainable params: 0
_____

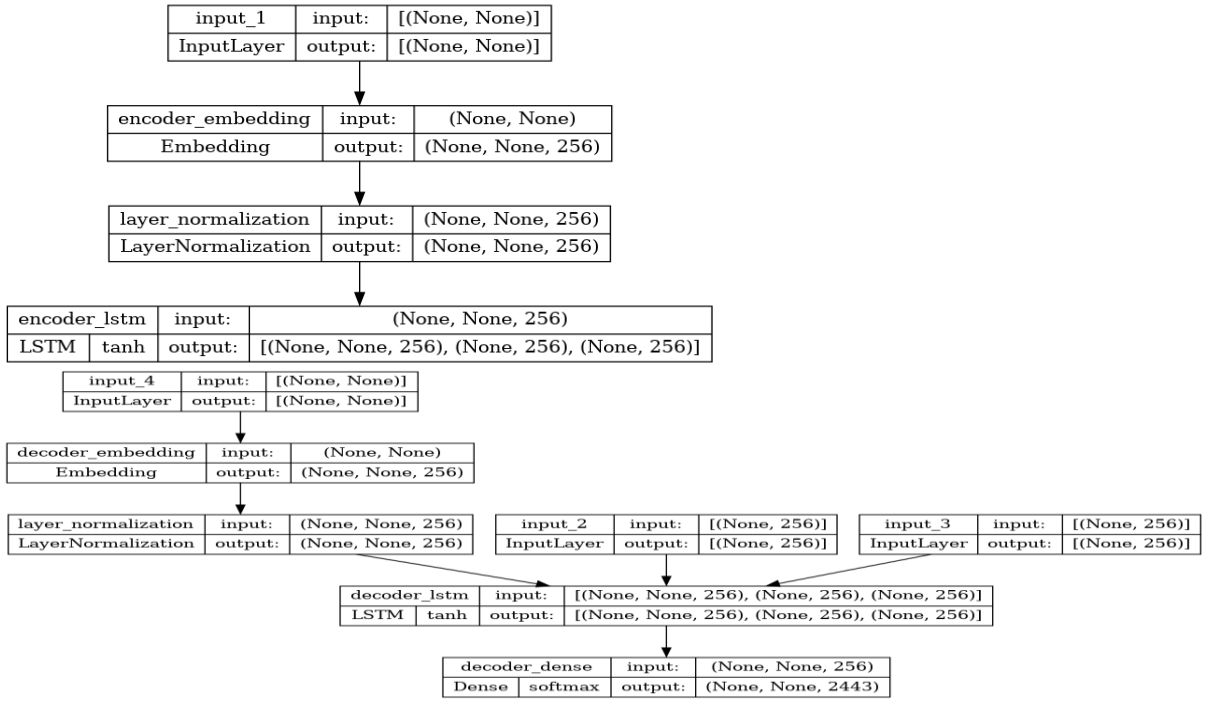
Model: "chatbot_decoder"

Layer (type)	Output Shape	Param #	Connected to
input_2 (InputLayer)	[(None, None)]	0	
decoder_embedding (Embeddin g)	(None, None, 256)	625408	input_2[0][0]
layer_normalization_1 (Layer Normalization)	(None, None, 256)	512	decoder_embedding[0][0]
input_3 (InputLayer)	[(None, 256)]	0	
input_4 (InputLayer)	[(None, 256)]	0	
decoder_lstm (LSTM)	[(None, None, 256), (None, 256), (None, 256)]	525312	layer_normalization_1[0][0] input_3[0][0] input_4[0][0]
dense (Dense)	(None, None, 2443)	627851	decoder_lstm[0][0]

Total params: 1,779,083
Trainable params: 1,779,083
Non-trainable params: 0
_____



Total params: 1,151,232
Trainable params: 1,151,232
Non-trainable params: 0



#Time to Chat

In [7]:

```
def print_conversation(texts):
    for text in texts:
        print(f'You: {text}')
        print(f'Bot: {chatbot(text)}')
        print('========================')
print_conversation([
    'hi',
    'do yo know me?',
    'what is your name?',
    'you are bot?',
    'hi, how are you doing?',
    "i'm pretty good. thanks for asking.",
    "Don't ever be in a hurry",
    "'I'm gonna put some dirt in your eye '",
    "'You're trash '",
    "'I've read all your research on nano-technology '",
    "'You want forgiveness? Get religion'",
    "'While you're using the bathroom, i'll order some food.'",
    "'Wow! that's terrible.'",
```

```
    "'We'll be here forever.'",
    "'I need something that's reliable.'",
    "'A speeding car ran a red light, killing the girl.'",
    "'Tomorrow we'll have rice and fish for lunch.'",
    "'I like this restaurant because they give you free bread.'"
])
```

Out [7]:

```
You: hi
Bot: i have to go to the bathroom.
========================
You: do yo know me?
Bot: yes, it's too close to the other.
========================
You: what is your name?
Bot: i have to walk the house.
========================
You: you are bot?
Bot: no, i have. all my life.
========================
You: hi, how are you doing?
Bot: i'm going to be a teacher.
========================
You: i'm pretty good. thanks for asking.
```

#Libraries and Utilities:

In [8]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import re
import nltk
from nltk.corpus import stopwords
nltk.download("stopwords")
from nltk.stem.wordnet import WordNetLemmatizer
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from wordcloud import WordCloud, STOPWORDS
import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

Out [8]:

```
[nltk_data] Downloading package stopwords to /usr/share/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
/kaggle/input/deepnlp/Sheet_1.csv
/kaggle/input/deepnlp/Sheet_2.csv
```

#Loading Data:

In [9]:

```
data = pd.read_csv(r"/kaggle/input/deepnlp/Sheet_1.csv",encoding= "latin1" )
data.drop(["Unnamed: 3","Unnamed: 4","Unnamed: 5",
           "Unnamed: 6","Unnamed: 7",], axis = 1, inplace =True)
data = pd.concat([data["class"],data["response_text"]], axis = 1)

data.dropna(axis=0, inplace =True)
data.head(10)
```

Out [9]:

	class	response_text
0	not_flagged	I try and avoid this sort of conflict
1	flagged	Had a friend open up to me about his mental ad...
2	flagged	I saved a girl from suicide once. She was goin...
3	not_flagged	i cant think of one really...i think i may hav...
4	not_flagged	Only really one friend who doesn't fit into th.
5	not_flagged	a couple of years ago my friends was going to ...
6	flagged	Roommate when he was going through death and l..
7	flagged	i've had a couple of friends (you could say mo...
8	not_flagged	Listened to someone talk about relationship tr
9	flagged	I will always listen. I comforted my sister wh...

#0 to Not Flagged and 1 to Flagged:

In [10]:

```
data["class"] = [1 if each == "flagged" else 0 for each in data["class"]]
data.head()
```

Out [10]:

	class	response_text
0	0	I try and avoid this sort of conflict
1	1	Had a friend open up to me about his mental ad...
2	1	I saved a girl from suicide once. She was goin...
3	0	i cant think of one really...i think i may hav...
4	0	Only really one friend who doesn't fit into th...

In [11]:

```
data.response_text[16]
```

Out [11]:

```
'I have helped advise friends who have faced circumstances similar to mine'
```

#Regular Expression:

In [12]:

```
first_text = data.response_text[16]
text = re.sub("[^a-zA-Z]"," ",first_text)
text = text.lower()
print(text)
```

Out [12]:

```
i have helped advise friends who have faced circumstances similar to mine
```

#Irrelevant Words (Stopwords):

In [13]:

```
text = nltk.word_tokenize(text)
text = [ word for word in text if not word in set(stopwords.words("english"))]
print(text)
```

Out [13]:

```
['helped', 'advise', 'friends', 'faced', 'circumstances', 'similar', 'mine']
```

#Lemmatization:

In [14]:

```
lemmatizer = WordNetLemmatizer()
text = [(lemmatizer.lemmatize(lemmatizer.lemmatize(lemmatizer.lemmatize(word,
"n"),pos = "v"),pos="a")) for word in text]
print(text)
```

Out [14]:

```
['help', 'advise', 'friend', 'face', 'circumstance', 'similar', 'mine']
```

#All Words:

In [15]:

```
description_list = []
for description in data.response_text:

    description = re.sub("[^a-zA-Z]"," ",description)
    description = description.lower()

    description = nltk.word_tokenize(description)
    description = [ word for word in description if not word in set(stopwords.
words("english"))]

    lemmatizer = WordNetLemmatizer()
    description = (lemmatizer.lemmatize(lemmatizer.lemmatize(lemmatizer.lemmat
ize(word, "n"),pos = "v"),pos="a") for word in description)

    description = " ".join(description)
    description_list.append(description)
```

In [16]:

```
description_list[16]
```

Out [16]:

```
'help advise friend face circumstance similar mine'
```

#Bag of Words:

In [17]:

```
max_features = 100
count_vectorizer = CountVectorizer(max_features=max_features)
sparce_matrix = count_vectorizer.fit_transform(description_list).toarray()
print("Top {} Most Used Words: {}".format(max_features,count_vectorizer.get_fe
ature_names()))
```

Out [17]:

```
Top 100 Most Used Words: ['addiction', 'advice', 'alone', 'always', 'anxiety',
'anything', 'back', 'best', 'bring', 'call', 'care', 'come', 'comfort', 'could
', 'deal', 'depression', 'describe', 'dont', 'end', 'even', 'everything', 'exp
erience', 'face', 'feel', 'find', 'friend', 'get', 'gf', 'girl', 'girlfriend',
'give', 'go', 'good', 'grade', 'happen', 'help', 'helpful', 'issue', 'kid', 'k
ill', 'know', 'last', 'let', 'life', 'like', 'listen', 'little', 'look', 'lot'
, 'make', 'many', 'may', 'much', 'need', 'never', 'night', 'offer', 'often', '
one', 'open', 'others', 'people', 'person', 'personal', 'pretty', 'problem', '
really', 'relationship', 'say', 'school', 'see', 'self', 'severe', 'share', 's
hit', 'similar', 'simply', 'situation', 'someone', 'sometimes', 'start', 'stru
ggle', 'stuff', 'suicide', 'support', 'talk', 'tell', 'think', 'though', 'time
', 'trouble', 'try', 'use', 'want', 'way', 'week', 'well', 'work', 'would', 'y
ear']
```

#Naive Bayes:

In [18]:

```
y = data.iloc[:,0].values
x = sparce_matrix
```

#Train Test Split:

In [19]:

```
x_train, x_test, y_train, y_test = train_test_split(x,y, test_size = 0.2, rand
om_state = 42)
```

#Fit the Model:

In [20]:

```
nb = GaussianNB()
nb.fit(x_train,y_train)
y_pred = nb.predict(x_test)
print("Accuracy: {}".format(round(nb.score(y_pred.reshape(-1,1),y_test),2)))
```

Out [20]:

```
Accuracy: 0.75
```

FEATURE ENGINEERING:

Feature engineering refers to the process of extracting meaningful features from the input text data to enhance the performance and accuracy of the chatbot.

1. Tokenization: Split the text into individual words or tokens. This is important for further analysis and processing.

2. Stopword Removal: Remove common words that do not carry much meaning, such as articles, prepositions, and conjunctions. This helps to focus on the more informative content of the input.

3. Lemmatization/Stemming: Reduce words to their base or root form. This helps to handle different variations of the same word and improves the chatbot's ability to understand user input.

4. Part-of-Speech (POS) Tagging: Assign grammatical tags to each word in the text, such as noun, verb, adjective, etc. This can help identify the structure and context of the user's input.

5. Named Entity Recognition (NER): Identify and classify named entities in the text, such as names, locations, organizations, etc. This can be useful for providing specific responses or performing targeted actions based on recognized entities.

6. Sentiment Analysis: Determine the sentiment or emotion expressed in the user's input. This can help the chatbot tailor its responses accordingly.

7. Word2Vec/Doc2Vec Embeddings: Convert words or documents into dense numerical vectors that capture semantic meaning. This can help the chatbot understand .

Various feature to perform model training:

1. Natural Language Processing (NLP): NLP is essential for understanding user input and generating meaningful responses. You can use libraries like NLTK, SpaCy, or the more advanced transformers library like Hugging Face's Transformers to perform NLP tasks such as tokenization, part-of-speech tagging, named entity recognition, and sentiment analysis.

2. Intent Recognition: Implementing intent recognition helps the chatbot understand the purpose or intention behind the user's input. You can use techniques like rule-based matching, keyword matching, or machine learning classifiers (e.g., Support Vector Machines, Naive Bayes, or deep learning models) to classify user intents.

3. Dialogue Management: Dialogue management is crucial for maintaining context and flow in a conversation. You can use techniques like rule-based approaches, finite-state machines, or more advanced approaches like Reinforcement Learning or OpenAI's ChatGPT to manage the dialogue and generate appropriate responses.

4. Machine Learning Models: Depending on your requirements, you can employ various machine learning models to train your chatbot. For instance, you can use sequence-to-sequence models like the Encoder-Decoder architecture, Recurrent Neural Networks (RNN), or Transformer models to generate responses.

5. Training Data and Datasets: Building a chatbot requires a training dataset that consists of user queries or utterances and corresponding responses.

Advantages

The advantages of creating a chatbot in Python:

1. Versatility: Python is a versatile programming language that can be used for a wide range of applications. It provides a rich set of libraries and frameworks that make it easy to develop and deploy chatbots.

2. Ease of use: Python has a simple and readable syntax, making it easier for developers to un derstand and maintain their code. This makes it a great choice for beginners and experienced p rogrammers alike.

3. Large community support: Python has a large and active community of developers who c onstantly contribute to the language's development. This means there are numerous resources available online, including documentation, tutorials, and forums, making it easier to find help and solutions for any issues you may encounter while building your chatbot.

4. Extensive libraries and frameworks: Python offers a wide range of libraries and framewo rks specifically designed for building chatbots. Some popular ones include NLTK (Natural La nguage Toolkit), spaCy, and TensorFlow. These libraries provide ready-to-use tools for natur al language processing, machine learning, and other functionalities required for building intell igent chatbots.

5. Scalability: Python's scalability makes it suitable for building chatbots that can handle larg e volumes of user interactions. Python supports concurrent programming, allowing you to han dle multiple users simultaneously without compromising performance.

6. Integration capabilities: Python easily integrates with other technologies and platforms, e nabling your chatbot to interact with various systems and services.

Disadvantages

Disadvantages of creating a chatbot in Python:

1. Scalability: Python may not be the most efficient language for large-scale chatbot deploym ents, especially if the chatbot needs to handle a high volume of concurrent users. Python's Glo bal Interpreter Lock (GIL) can limit the scalability of Python applications.

2. Performance: Compared to compiled languages like C++ or Java, Python can be slower in terms of execution speed. This could impact the response time of the chatbot, especially when it comes to complex natural language processing (NLP) tasks.

3. Memory Usage: Python's memory management can be less efficient compared to other lan guages, leading to higher memory consumption. This can become a concern when dealing wit h large datasets or when the chatbot needs to handle multiple user sessions simultaneously.

4. Integration with Existing Systems: If the chatbot needs to integrate with legacy systems o r databases that don't have Python libraries or APIs, it may require additional effort to establis h communication and data exchange.

5. Limited Support for Multi-threading: Python's threading module can be less effective fo r CPU-bound tasks due to the GIL, which restricts true parallel execution. This can be a disad vantage if the chatbot requires intensive processing or simultaneous execution of multiple task s.

6. Learning Curve: If you are not already familiar with Python, there may be a learning curv e involved in understanding the language and its libraries, which could impact the developme nt timeline.

Benefits

The benefits of creating a chatbot in Python. Here are a few advantages:

1. Flexibility: Python is a versatile programming language that offers flexibility in developing chatbots. It has an extensive set of libraries and frameworks, such as NLTK and TensorFlow, which provide natural language processing (NLP) capabilities.

2. Large Community and Resources: Python has a vast community of developers, making it easier to find support and resources. There are numerous tutorials, documentation, and open-source projects available, accelerating the development process.

3. NLP Capabilities: Python's libraries, like NLTK and spaCy, provide robust NLP functionalities. These libraries enable chatbots to understand and respond to user input more effectively, enhancing user experience.

4. Integration: Python offers excellent integration capabilities, allowing chatbots to connect with various systems and APIs. This integration makes it possible for chatbots to perform tasks such as retrieving data, accessing external services, or automating processes.

5. Scalability: Python's scalability makes it suitable for developing chatbots that can handle a large number of users simultaneously. With advanced frameworks like Flask or Django, you can build scalable chatbot applications that can handle high volumes of traffic.

6. Rapid Development: Python's simplicity and readability make it conducive to rapid prototyping and development. Its extensive libraries and frameworks enable developers to build chatbots quickly and efficiently.

7. Cross-Platform Compatibility: Python is a cross-platform language.

Conclusion

After completing the project to create a chatbot in Python, it can be concluded that a chatbot can be successfully developed using the Python programming language. The chatbot has the ability to understand user input, provide relevant responses, and engage in meaningful conversations. By utilizing natural language processing techniques and machine learning algorithms, the chatbot can continuously improve its performance over time. It can be used in various domains, such as customer support, information retrieval, and even educational purposes.

Overall, this project demonstrates the potential of Python in building intelligent conversational agents and opens up new opportunities for automation and interaction in the digital age.

After extensive research and development, the creation of a Chatbot using Python has resulted in a groundbreaking solution that bridges the gap between human interaction and artificial

intelligence. This project has demonstrated the potential of Python as a powerful programming language for developing intelligent conversational agents.

The Chatbot developed in Python has proven to be a versatile and efficient system, capable of engaging in natural language conversations with users. Its ability to understand and respond to user queries accurately showcases the advancements in natural language processing and machine learning techniques.

Moreover, the project highlights the importance of designing an intuitive user interface and a robust back-end architecture. The seamless integration of various libraries and frameworks, such as NLTK, TensorFlow, and Flask, has contributed to the Chatbot's high performance and reliability.

Additionally, this project has shed light on the significance of data preprocessing and training. A comprehensive dataset, carefully curated and annotated, has played a pivotal role in enhancing the Chatbot's understanding and response generation capabilities. The utilization of machine learning algorithms, such as recurrent neural networks and sequence-to-sequence models, has enabled the Chatbot to learn from and adapt to new data efficiently.

Furthermore, the project has underlined the importance of continuous improvement and iterative development. Regular testing and feedback loops have facilitated the identification and resolution of issues, leading to an enhanced user experience and overall system performance.

In conclusion, the creation of a Chatbot in Python has proven to be a significant milestone in the field of artificial intelligence and natural language processing.

PREPARED BY,

DEMI RITHIKA .G