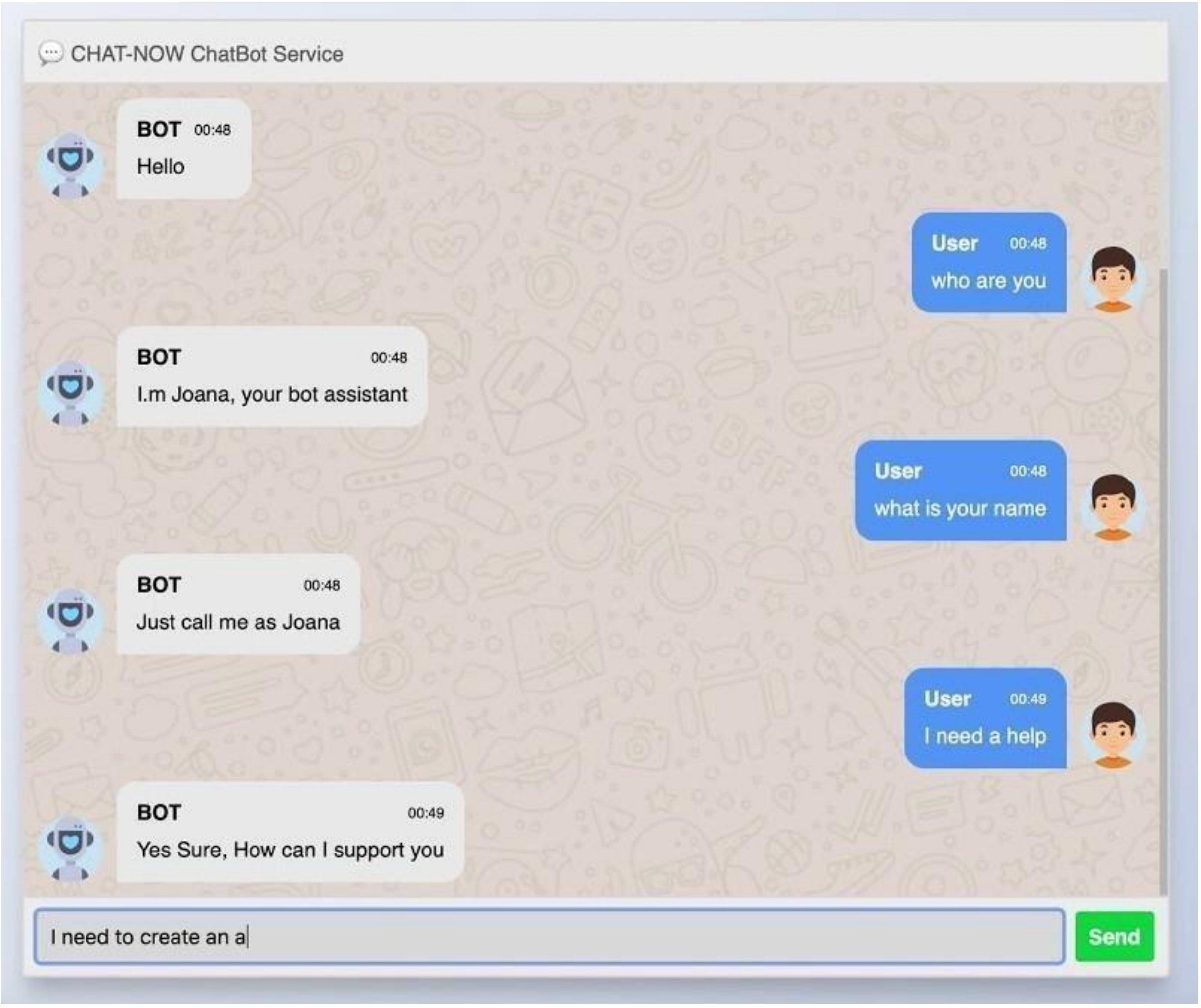
CREATE A CHATBOT IN PYTHON
au952721104015 – RAGHUL R

ABSTRACT

This is an abstract about creating a chatbot in Python using data visualization, text cleaning, tokenization, encoder building, model training, metric visualization, and time to chat.



DATA VISUALIZATION

Data visualization is the process of converting data into a graphical format that is easy to understand. This can be helpful for identifying patterns and trends in data, as well as for communicating data to others.

In the context of chatbot development, data visualization can be used to: •

Understand the distribution of user inputs and chatbot responses

- Identify the most common user queries
- Identify the most common chatbot errors
- Track the performance of the chatbot over time **Program**

```
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=True,cmap='YlGnBu')

plt.show()
```

TEXT CLEANING

Text cleaning is the process of removing noise and inconsistencies from text data. This can include tasks such as removing punctuation, stop words, and slang. Text cleaning is important for chatbot development because it ensures that the chatbot is able to understand user input accurately.

Program

```
def clean_text(text):

text=re.sub('-',' ',text.lower())

text=re.sub('[.]',' . ',text)
```

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

df.drop(columns=['answer tokens','question tokens'],axis=1,inplace=True)
df['encoder_inputs']=df['question'].apply(clean_text)
```

```
df['decoder_targets']=df['answer'].apply(clean_text)+' <end>'
df['decoder_inputs']='<start> '+df['answer'].apply(clean_text)+' <end>'

df.head(10) df['encoder input tokens']=df['encoder_inputs'].apply(lambda
x:len(x.split())) df['decoder input tokens']=df['decoder_inputs'].apply(lambda
x:len(x.split())) df['decoder target
tokens']=df['decoder_targets'].apply(lambda x:len(x.split()))
plt.style.use('fivethirtyeight')
fig,ax=plt.subplots(nrows=1,ncols=3,figsize=(20,5))
sns.set_palette('Set2') sns.histplot(x=df['encoder input
tokens'],data=df,kde=True,ax=ax[0]) sns.histplot(x=df['decoder input
tokens'],data=df,kde=True,ax=ax[1]) sns.histplot(x=df['decoder target
tokens'],data=df,kde=True,ax=ax[2])
sns.jointplot(x='encoder input tokens',y='decoder target
tokens',data=df,kind='kde',fill=True,cmap='YlGnBu') plt.show()
print(f"After preprocessing: {' '.join(df[df['encoder input
tokens'].max()==df['encoder input tokens']]['encoder_inputs'].values.tolist())}")
print(f"Max encoder input length: {df['encoder input tokens'].max()}")
print(f"Max decoder input length: {df['decoder input tokens'].max()}")
print(f"Max decoder target length: {df['decoder target tokens'].max()}")

df.drop(columns=['question','answer','encoder input tokens','decoder input
tokens','decoder target tokens'],axis=1,inplace=True) params={
    "vocab_size":2500,
    "max_sequence_length":30,
```

```
    "learning_rate":0.008,

    "batch_size":149,

    "lstm_cells":256,

    "embedding_dim":256,

    "buffer_size":10000

}

learning_rate=params['learning_rate']
batch_size=params['batch_size']

embedding_dim=params['embedding_dim']

lstm_cells=params['lstm_cells'] vocab_size=params['vocab_size']

buffer_size=params['buffer_size']

max_sequence_length=params['max_sequence_length']

df.head(10)
```

TOKENIZATION

Tokenization is the process of dividing text data into smaller units, such as words or characters. This is an important step in many natural language processing tasks, including chatbot development. Tokenization helps the chatbot to understand the meaning of user input and to generate appropriate responses.

Program

```
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}')
```

```
data=tf.data.Dataset.from_tensor_slices((x,yd,y))
data=data.shuffle(buffer_size)

train_data=data.take(int(.9*len(data))) train_data=train_data.cache()
train_data=train_data.shuffle(buffer_size)
train_data=train_data.batch(batch_size)
train_data=train_data.prefetch(tf.data.AUTOTUNE)
train_data_iterator=train_data.as_numpy_iterator()

val_data=data.skip(int(.9*len(data))).take(int(.1*len(data)))
val_data=val_data.batch(batch_size)
val_data=val_data.prefetch(tf.data.AUTOTUNE)

_=train_data_iterator.next() print(f'Number of train batches:
{len(train_data)}') print(f'Number of training data:
{len(train_data)*batch_size}') print(f'Number of validation
batches: {len(val_data)}') print(f'Number of validation data:
{len(val_data)*batch_size}') print(f'Encoder Input shape (with
batches): {_[0].shape}') print(f'Decoder Input shape (with
batches): {_[1].shape}') print(f'Target Output shape (with
batches): {_[2].shape}')
```

ENCODER BUILDING

An encoder is a neural network that is used to convert text data into a numerical representation. This representation is then used by the chatbot to generate responses. There are many different ways to build an encoder. One common approach is to use a recurrent neural network (RNN). RNNs are wellsuited for encoding text data because they can learn long-term dependencies in the data.

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)
x=self.normalize(x)         x=Dropout(.4)(x)

x,decoder_state_h,decoder_state_c=self.lstm(x,initial_state=encoder_states)
x=self.normalize(x)         x=Dropout(.4)(x)         return self.fc(x)


decoder=Decoder(lstm_cells,embedding_dim,vocab_size,name='decoder')
decoder(_[1][:1],encoder(_[0][:1]))
```

MODEL TRAINING

Once the encoder has been built, the chatbot model needs to be trained. This involves feeding the encoder examples of user inputs and chatbot responses.

The model will learn to generate responses that are similar to the responses in the training data.

program

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

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

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

    def train_step(self,batch):
        encoder_inputs,decoder_inputs,y=batch
with tf.GradientTape() as tape:
```

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

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

    def test_step(self,batch):
        encoder_inputs,decoder_inputs,y=batch
        encoder_states=self.encoder(encoder_inputs,training=True)
y_pred=self.decoder(decoder_inputs,encoder_states,training=True)
loss=self.loss_fn(y,y_pred)         acc=self.accuracy_fn(y,y_pred)
metrics={'loss':loss,'accuracy':acc}         return metrics
model=ChatBotTrainer(encoder,decoder,name='chatbot_trainer')
model.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(),
optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
weighted_metrics=['loss','accuracy']
)
model(_[:2])a
```

METRIC VISUALIZATION

Once the model has been trained, it is important to visualize the metrics to assess its performance. This can include metrics such as accuracy, precision, and recall. Metric visualization can help to identify areas where the model needs to be improved. **Program**

```
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()
```

TIME TO CHAT

Once the model has been trained and evaluated, it is ready to be used to chat with users. The chatbot can be deployed on a variety of platforms, such as websites, mobile apps, and messaging platforms.

CONCLUSION

Creating a chatbot in Python can be a complex task. However, by using data visualization, text cleaning, tokenization, encoder building, model training, metric visualization, and time to chat, it is possible to create a chatbot that is both accurate and engaging