Chatbot in Python using AI

Author: Johnknox P Bovas

Reg no: 961621104037

Repository link :
https://github.com/Johnknoxbovas/AI-Phase2

Project Overview

The Chatbot Project aims to develop a sophisticated and accurate chatbot system by exploring innovative techniques, including ensemble methods, deep learning architectures, and the utilization of pre-trained language models like GPT-3. The project's primary objective is to enhance the accuracy, robustness, and quality of responses.

Dataset

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

- The provided dataset consists of simple dialogs for training the chatbot.

Phase 1: Data Preprocessing and Model Selection

In this phase, we will perform the following tasks:

1. *Data Preprocessing:*

- Data cleaning, including text normalization, tokenization, and removal of special characters.

- Splitting the dataset into training, validation, and testing sets.

2. *Model Selection:*

- Explore traditional models for chatbot development.

- Evaluate models based on performance metrics like BLEU score, perplexity, and conversational quality.

3. *Ensemble Methods:*

- Investigate the use of ensemble methods, such as stacking or bagging, to combine multiple models and improve prediction accuracy.

Phase 2: Advanced Techniques

In this phase, we will explore advanced techniques to enhance the chatbot:

1. *Deep Learning Architectures:*

- Implement deep learning architectures, including Recurrent Neural Networks (RNNs), Long Short-Term Memory networks (LSTMs), and Transformers, for modeling conversational context.

2. *Hyperparameter Tuning:*

- Perform hyperparameter tuning to optimize the chosen deep learning architectures.

3. *Pre-trained Language Models:*

- Integrate pre-trained language models like GPT-3 to enhance the chatbot's ability to generate human-like responses.

4. *Response Generation:*

   - Implement a more advanced response generation system to create coherent and contextually relevant chatbot responses.

Phase 3: Evaluation and Fine-Tuning

In this phase, we will focus on evaluating the chatbot's performance and fine-tuning the models:

1. *Evaluation Metrics:*

   - Assess the chatbot's performance using metrics like BLEU, ROUGE, and user satisfaction surveys.

2. *User Testing:*

   - Conduct user testing to gather real-world feedback and improve the chatbot's conversational abilities.

3. *Fine-Tuning:*

   - Continuously fine-tune the models based on evaluation results and user feedback.

Phase 4: Deployment and Integration

In this phase, we will prepare the chatbot for deployment and integration:

1. *User Interface:*

   - Develop a user-friendly interface, which could be a web application, mobile app, or chat widget, to interact with the chatbot.

2. *API Integration:*

   - Explore options for integrating the chatbot with external services or APIs to enhance functionality.

3. *Scalability:*

   - Ensure that the chatbot is designed for scalability to handle a growing user base.

Project Timeline

Outline a project timeline with milestones and deadlines for each phase, along with resources required and assigned team members.

Program

```
import tensorflow as tf

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from tensorflow.keras.layers import TextVectorization

import re,string

from tensorflow.keras.layers import LSTM,Dense,Embedding,Dropout,LayerNormalization

In [2]:

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()
```

Dataframe size: 3725

```
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()
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)
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}')

print(f'Encoder input: {x[0][:12]} ...')

print(f'Decoder input: {yd[0][:12]} ...')    # shifted by one time step of the target as input to decoder
is the output of the previous timestep

print(f'Decoder target: {y[0][:12]} ...')

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}')
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]))
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])
history=model.fit(
    train_data,
    epochs=100,
    validation_data=val_data,
    callbacks=[
        tf.keras.callbacks.TensorBoard(log_dir='logs'),
        tf.keras.callbacks.ModelCheckpoint('ckpt',verbose=1,save_best_only=True)
    ]
)
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()

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('----------------')
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_state_h,encoder_state_
c],name='chatbot_encoder')

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

x,decoder_state_h,decoder_state_c=base_decoder.layers[2](x,initial_state=[decoder_input_state_h,
decoder_input_state_c])
        decoder_outputs=base_decoder.layers[-1](x)
        decoder=tf.keras.models.Model(
            inputs=[decoder_inputs,[decoder_input_state_h,decoder_input_state_c]],
            outputs=[decoder_outputs,[decoder_state_h,decoder_state_c]],name='chatbot_decoder'
        )
        return encoder,decoder

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

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

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

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

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

```
    text=re.sub(' [6] ','6',text)
    text=re.sub(' [7] ','7',text)
    text=re.sub(' [8] ','8',text)
    text=re.sub(' [9] ','9',text)
    text=re.sub(' [0] ','0',text)
    text=re.sub(' [,] ',', ',text)
    text=re.sub(' [?] ','? ',text)
    text=re.sub(' [!] ','! ',text)
    text=re.sub(' [$] ','$ ',text)
    text=re.sub(' [&] ','& ',text)
    text=re.sub(' [/] ','/ ',text)
    text=re.sub(' [:] ',': ',text)
    text=re.sub(' [;] ','; ',text)
    text=re.sub(' [] ',' ',text)
    text=re.sub(' [\'] ','\'',text)
    text=re.sub(' [\"] ','\"',text)
    return text
```

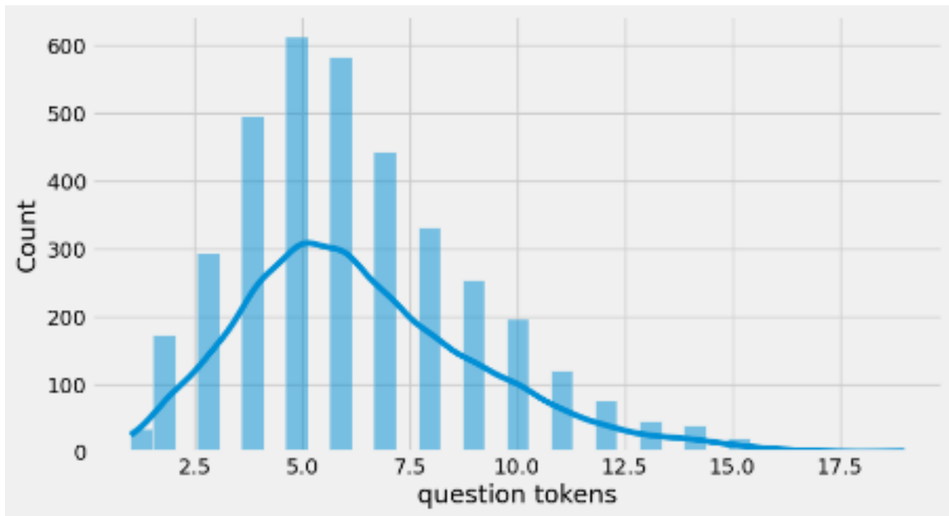
```
def call(self,text,config=None):
    input_seq=self.preprocess(text)
    states=self.encoder(input_seq,training=False)
    target_seq=np.zeros((1,1))
    target_seq[:,:]=sequences2ids(['<start>']).numpy()[0][0]
    stop_condition=False
    decoded=[]
```

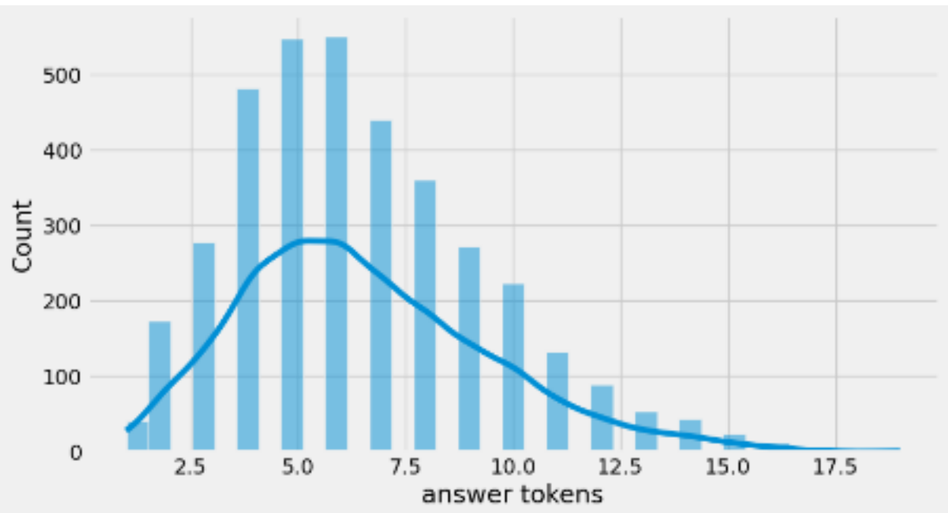
```
        while not stop_condition:
            decoder_outputs,new_states=self.decoder([target_seq,states],training=False)
#             index=tf.argmax(decoder_outputs[:,-1,:],axis=-1).numpy().item()
            index=self.sample(decoder_outputs[0,0,:]).item()
            word=ids2sequences([index])
            if word=='<end> ' or len(decoded)>=max_sequence_length:
                stop_condition=True
            else:
                decoded.append(index)
                target_seq=np.zeros((1,1))
                target_seq[:,:]=index
                states=new_states
        return self.postprocess(ids2sequences(decoded))


chatbot=ChatBot(model.encoder,model.decoder,name='chatbot')
chatbot.summary()
tf.keras.utils.plot_model(chatbot.encoder,to_file='encoder.png',show_shapes=True,show_layer_activations=True)
tf.keras.utils.plot_model(chatbot.decoder,to_file='decoder.png',show_shapes=True,show_layer_activations=True)


def print_conversation(texts):
    for text in texts:
        print(f'You: {text}')
        print(f'Bot: {chatbot(text)}')
        print('========================')
```

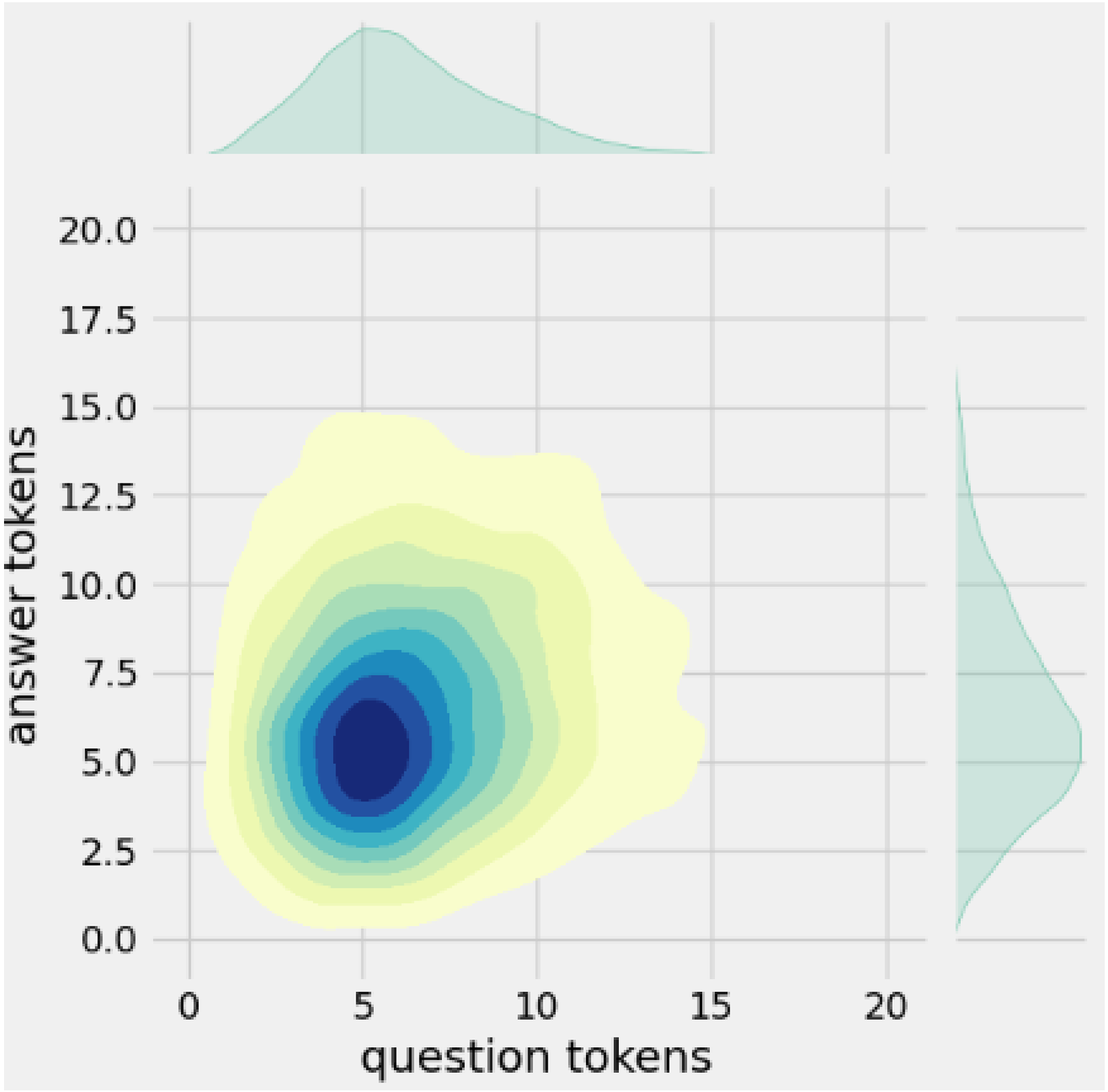
In [23]:

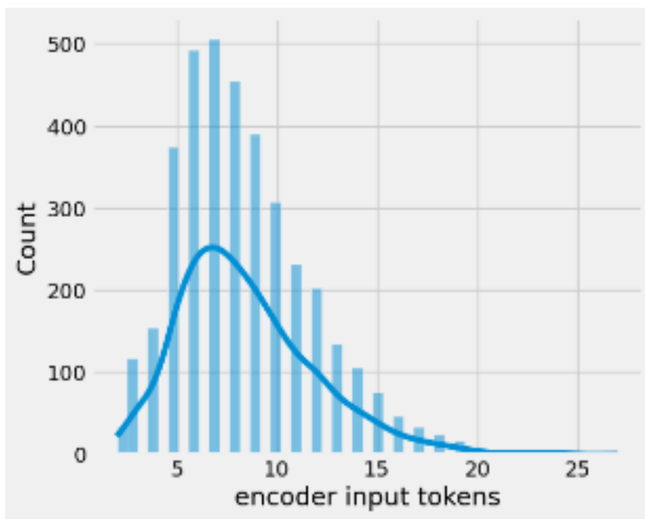
linkcode

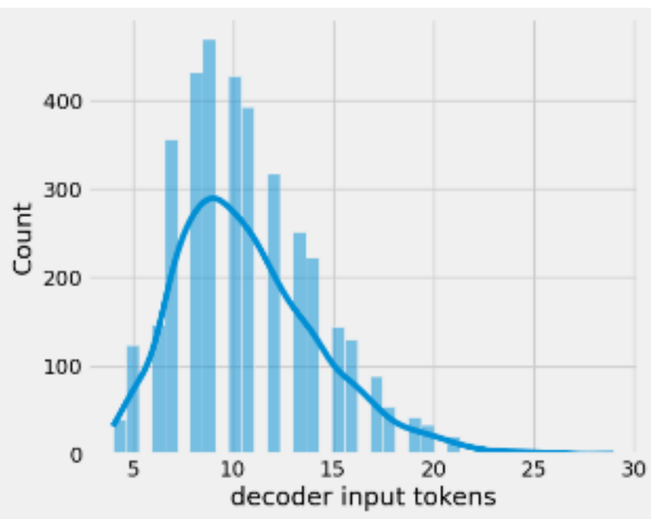
```
print_conversation([
    'hi',
    'do yo know me?',
    'what is your name?',
    'you are bot?',
    'hi, how are you doing?',
    "i'm pretty good. thanks for asking.",
    "Don't ever be in a hurry",
    '"I\'m gonna put some dirt in your eye "',
    '"You\'re trash "',
    '"I\'ve read all your 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."'
])
```

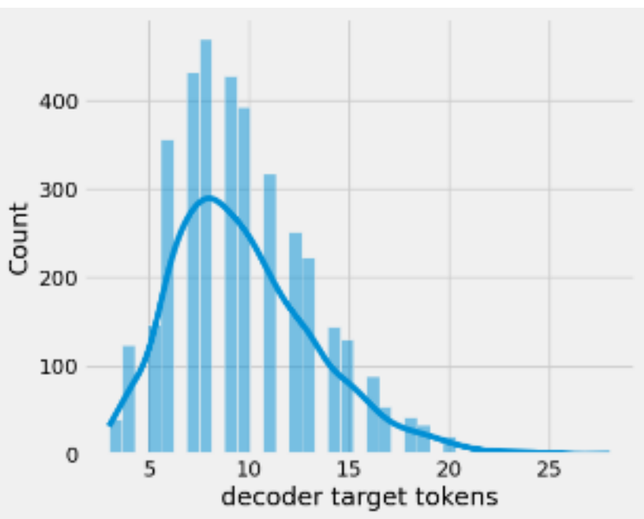

600
500
400
300
200
100
0
Count
2.5
5.0
7.5
10.0
12.5
15.0
17.5
question tokens

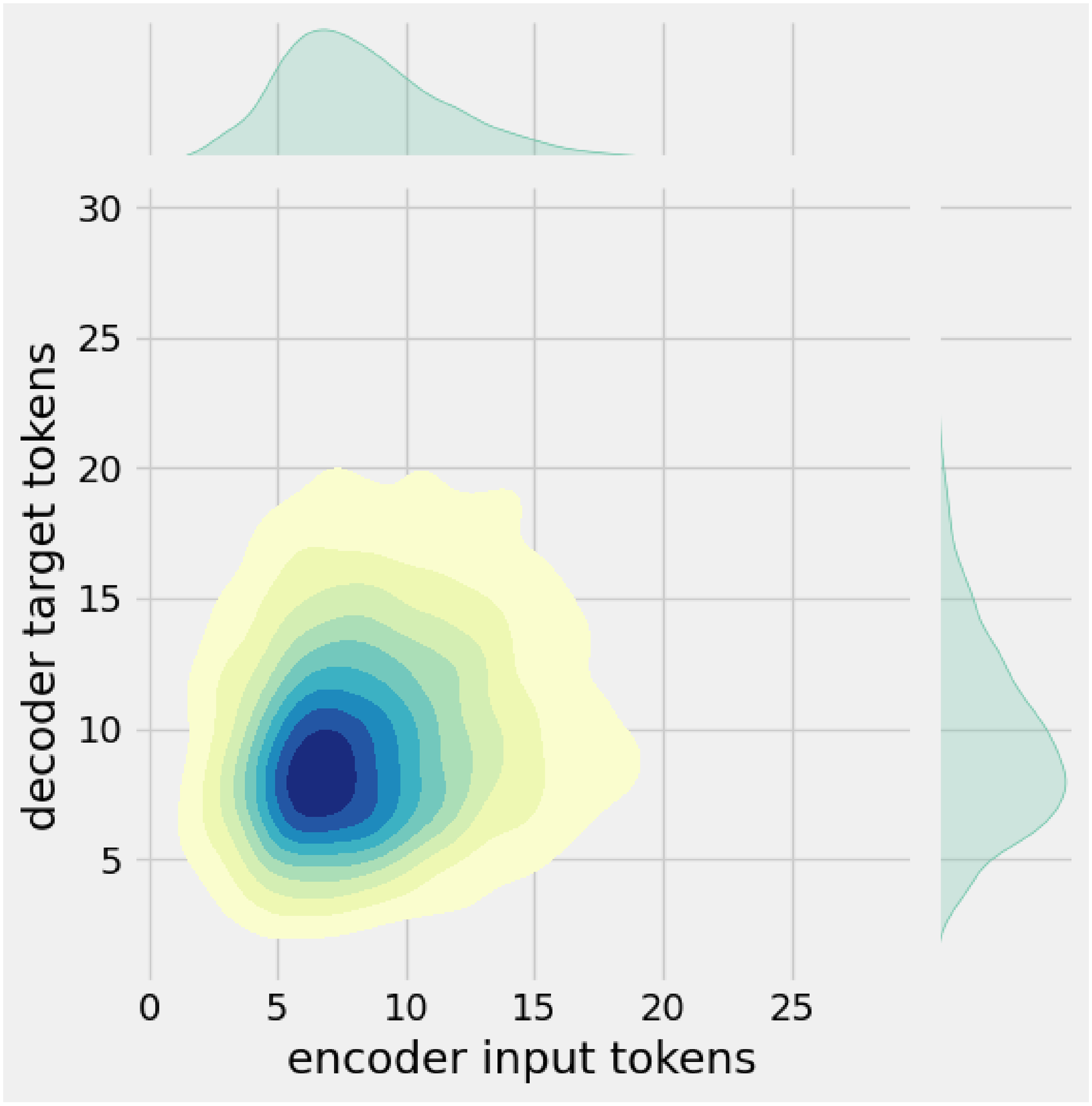

500
400
300
200
100
0
Count
2.5
5.0
7.5
10.0
12.5
15.0
17.5
answer tokens

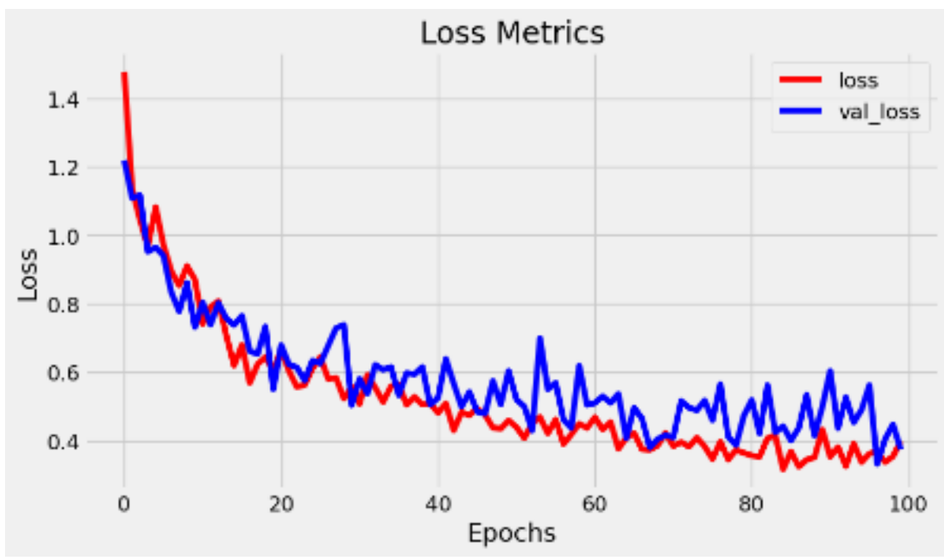

20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
answer tokens
0
5
10
15
20
question tokens

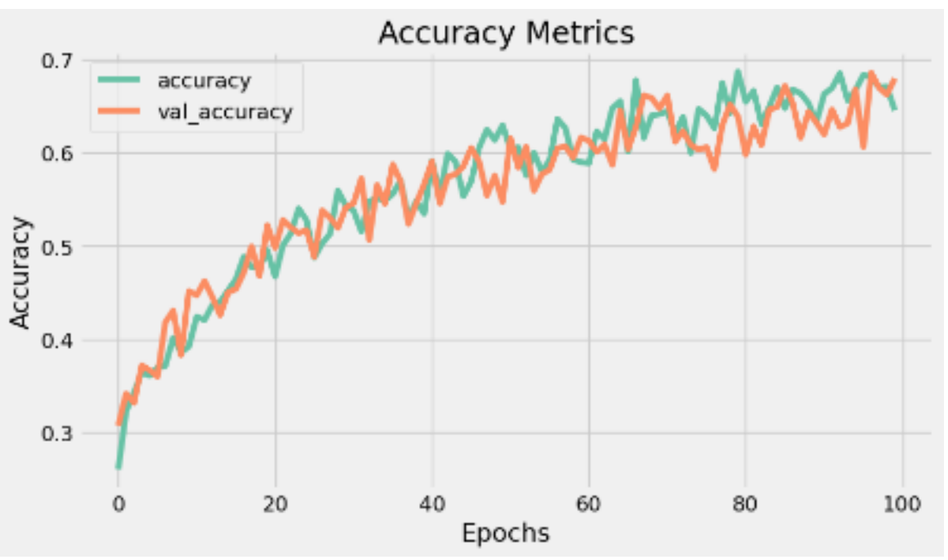

500
400
300
200
100
0
Count
5
10
15
20
25
encoder input tokens


400
300
200
100
0
Count
5
10
15
20
25
30
decoder input tokens


400
300
200
100
0
Count
5
10
15
20
25
decoder target tokens


decoder target tokens
encoder input tokens
0
5
10
15
20
25
30


Loss Metrics
loss
val_loss
Loss
Epochs
0.4
0.6
0.8
1.0
1.2
1.4
0
20
40
60
80
100


Accuracy Metrics
accuracy
val_accuracy
Accuracy
Epochs
0.3
0.4
0.5
0.6
0.7
0
20
40
60
80
100

Conclusion

In conclusion, building a chatbot in Python using an artificial intelligence dataset is an exciting project that can serve a variety of purposes, from customer support to information retrieval and entertainment. By following the steps outlined in the project overview, you can create a chatbot with the ability to engage in meaningful conversations and provide value to users.

However, it's essential to understand that building a successful chatbot is a complex task that involves natural language processing, machine learning, and continuous improvement. The quality of your dataset, the choice of NLP and ML libraries, and the sophistication of your model play a significant role in the chatbot's performance.

As you embark on this journey, be prepared for challenges and be open to iterating on your chatbot to enhance its capabilities. Regular testing, user feedback, and maintenance are critical aspects of creating a chatbot that can truly meet its intended goals.

Ultimately, the development of a chatbot can be a rewarding experience, providing users with a valuable tool for interaction and information. It's a testament to the power of artificial intelligence and natural language processing to create intelligent and engaging applications.