CREATE CHATBOT IN PYTHON

Au952721104011 – K MOHAMED MANSOOR

Phase-4 Submission

Project Title:create chatbot in python.

INTRODUCTION:

Creating a chatbot in Python is a fascinating and practical project that allows you to build an automated conversation partner. Chatbots are used in a wide range of applications, from customer support to virtual assistants and more. In this introduction, we'll outline the fundamental steps to create a basic chatbot in Python.

Step 1: Set Up Your Development Environment

Before you start building your chatbot, you need to have Python installed on your system. You can download and install Python from the [official website](https://www.python.org/). Additionally, you may want to use a code editor or integrated development environment (IDE) like Visual Studio Code, PyCharm, or Jupyter Notebook to write your Python code.

Step 2: Choose a Framework or Library

There are several Python libraries and frameworks available for building chatbots. Some popular ones include:

- **NLTK (Natural Language Toolkit):** NLTK is a comprehensive library for natural language processing. It provides tools for text processing and analysis.

- **spaCy:** spaCy is another natural language processing library known for its speed and accuracy. It can be used for various NLP tasks, including chatbot development.

- **TensorFlow and Keras:** These libraries are often used for building deep learning-based chatbots, such as sequence-to-sequence models.

- **Rasa:** Rasa is an open-source framework specifically designed for building conversational AI. It provides tools for intent recognition, dialogue management, and more.

- **ChatterBot:** ChatterBot is a simple machine learning-based library for creating chatbots. It's a good choice for beginners.

Choose a library or framework that suits your project's requirements and your level of expertise.

Step 3: Define the Chatbot's Purpose

Before you start coding, you should clearly define the purpose of your chatbot. What kind of conversations will it handle? Is it for customer support, general information, or entertainment? Understanding the chatbot's purpose will guide the development process.

Step 4: Create a Dataset

To train your chatbot, you'll need a dataset of sample conversations. This dataset should include both user inputs and corresponding chatbot responses. The quality and diversity of your dataset will greatly influence the chatbot's performance.

Step 5: Preprocess the Text

You'll need to preprocess the text in your dataset. This involves tasks like tokenization, removing stopwords, and converting text to lowercase to prepare the data for training and inference.

Step 6: Choose a Model

Depending on your chosen library or framework, you may need to select or design a chatbot model. This could be rule-based, machine learning-based, or deep learning-based, depending on the complexity of the chatbot.

Step 7: Train Your Chatbot

If you're using a machine learning-based approach, you'll need to train your chatbot using your dataset. This involves feeding the dataset into your chosen model and adjusting parameters for optimal performance.

Step 8: Implement User Interaction

Once your chatbot is trained, you can create a user interface for interacting with it. This could be a web-based interface, a command-line application, or an integration with an existing platform.

Step 9: Test and Refine

Test your chatbot thoroughly and collect user feedback. Refine the chatbot's responses and behavior based on the feedback and real-world usage.

Step 10: Deploy and Maintain

Finally, deploy your chatbot in a production environment if needed. Regularly maintain and update it to keep it relevant and effective.

Creating a chatbot is an iterative process, and you can continually improve and expand its capabilities as you gain experience. The above steps provide a high-level overview of the chatbot development process in Python. Depending on your project's complexity, you may need to dive deeper into NLP techniques, machine learning, or even deep learning to create a sophisticated conversational agent.

HERE the list of tools usung create chatbot in python

When creating a chatbot in Python, you can use a variety of tools and libraries to streamline the development process. Here are some of the commonly used tools and libraries for building chatbots in Python:

1. **Natural Language Processing (NLP) Libraries:**

   - **NLTK (Natural Language Toolkit):** NLTK is a comprehensive library for natural language processing. It provides tools for tokenization, stemming, lemmatization, part-of-speech tagging, and more.

   - **spaCy:** spaCy is another popular NLP library known for its speed and accuracy. It offers pre-trained models for various languages and NLP tasks.

   - **TextBlob:** TextBlob is a simplified NLP library that makes it easy to perform common NLP tasks like sentiment analysis, part-of-speech tagging, and translation.

2. **Machine Learning and Deep Learning Frameworks:**

   - **scikit-learn:** If you're building a rule-based or traditional machine learning chatbot, scikit-learn is a powerful library for classification and regression tasks.

   - **TensorFlow and Keras:** These libraries are used for building deep learning-based chatbots, such as sequence-to-sequence models and neural networks.

   - **PyTorch:** PyTorch is another popular deep learning framework that's widely used for natural language processing tasks.

3. **Chatbot Frameworks:**

   - **Rasa:** Rasa is an open-source framework designed specifically for building conversational AI applications. It provides tools for intent recognition, dialogue management, and more.

   - **ChatterBot:** ChatterBot is a simple machine learning-based library for creating chatbots. It's suitable for basic chatbot applications and is easy for beginners to get started with.

4. **Web Frameworks (for Chatbot Deployment):**

   - **Flask:** Flask is a lightweight web framework that you can use to create web-based chatbot interfaces.

   - **Django:** Django is a more comprehensive web framework that's suitable for building complex chatbot web applications.

5. **Cloud Services (for Hosting and Deployment):**

   - **Amazon Lex:** Amazon Lex is a cloud service provided by AWS for building conversational interfaces using automatic speech recognition and natural language understanding.

   - **Google Dialogflow:** Dialogflow is a cloud-based chatbot development platform by Google that offers natural language understanding and conversation management.

6. **Text-to-Speech (TTS) and Speech-to-Text (STT) Services:**

   - **Google Text-to-Speech API and Google Speech-to-Text API:** These services can be integrated into your chatbot to add voice interaction capabilities.

   - **IBM Watson Text to Speech and Speech to Text:** IBM Watson offers similar services for TTS and STT.

7. **Natural Language Understanding APIs:**

   - **IBM Watson NLU:** IBM Watson provides APIs for natural language understanding, sentiment analysis, and entity recognition.

   - **Google Natural Language API:** Google's NLU API can be used for sentiment analysis, entity recognition, and content classification.

8. **Bot Development Platforms:**

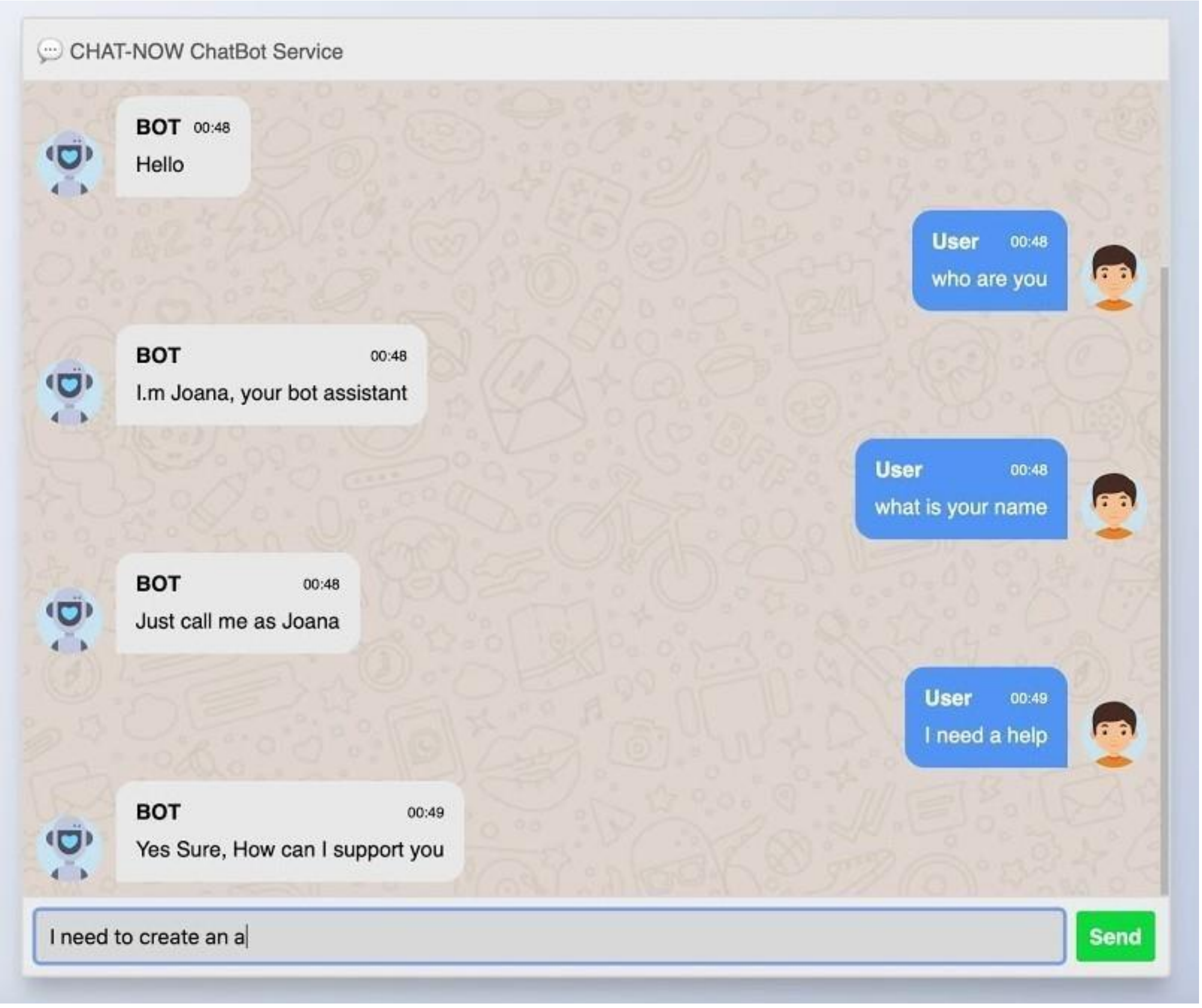
- **BotPress:** BotPress is an open-source bot development platform that allows you to build, deploy, and manage chatbots.

The choice of tools and libraries depends on the complexity and requirements of your chatbot project. For simple chatbots, you might use lightweight libraries, while more advanced chatbots may require deep learning frameworks and cloud-based services for hosting and deployment. Make sure to consider your project goals and select the tools that best fit your needs.

1.DESIGN THINKING

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

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.

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.

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.

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.

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.

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

2.INOVATION

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

3.Development part-1

Table Content

These three methods can greatly improve the NLU (Natural Language Understanding) classification training process in your chatbot development project and aid the preprocessing in text mining. Below we demonstrate how they can increase intent detection accuracy.

```
!git clone https://github.com/interds/3-methods-of-nlu-data-pre-processing.git
```

```
%cd ./3-methods-of-nlu-data-pre-processing
```

```
!apt-get install python3-venv
```

```
!python -m venv --system-site-packages ./venv
```

```
!source ./venv/bin/activate
```

```
!pip install rasa[transformers]
```

```
!pip install -U ipython # fix create_prompt_application
```

```
!pip install pandas
```

```
!pip install chatette
```

```
!pip install transformers
```

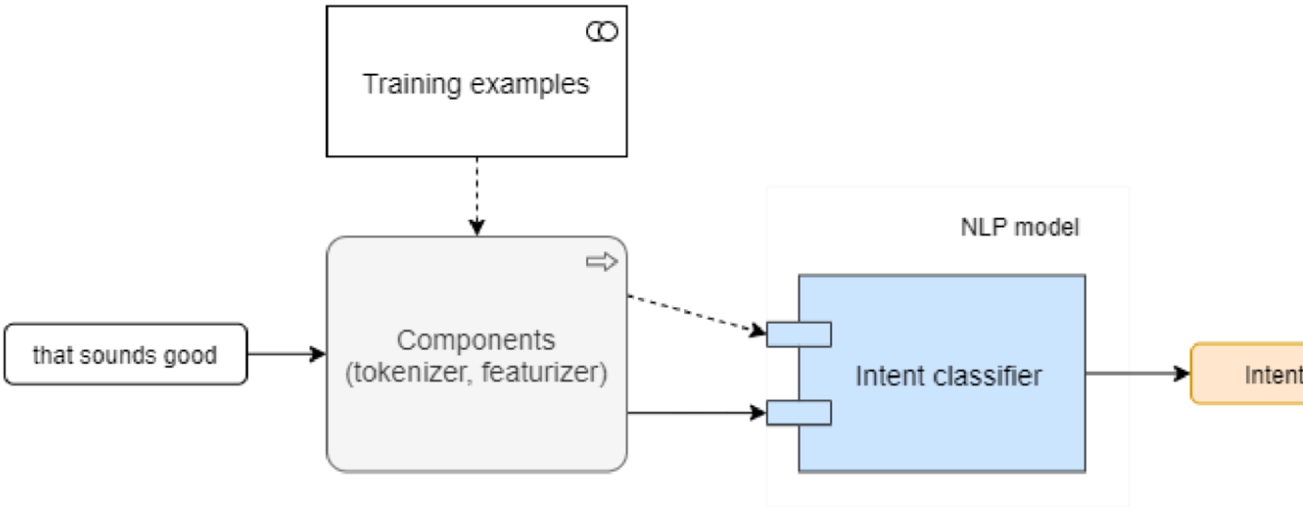
```
!pip install tensorflow_datasets
```

Initial model

Rasa's boilerplate generated by 'rasa init' is enough to demonstrate the initial model in our chatbot development effort.

We train and evaluate the model with the following config:

```
language: en
```




```
pipeline:
- name: WhitespaceTokenizer
- name: CountVectorsFeaturizer
- name: CountVectorsFeaturizer
  analyzer: "char_wb"
  min_ngram: 1
  max_ngram: 4
- name: DIETClassifier
  epochs: 100
```

```
!rasa train -c config-simple.yml --fixed-model-name simple --quiet
```

```
Training Core model...
```

```
2020-06-22 20:46:54.811928: E tensorflow/stream_executor/cuda/cuda_dr
iver.cc:351] failed call to cuInit: CUDA_ERROR_NO_DEVICE: no CUDA-cap
able device is detected
Core model training completed.

Training NLU model...

/usr/local/lib/python3.6/dist-packages/rasa/utils/common.py:363: User
Warning: You specified 'DIET' to train entities, but no entities are
present in the training data. Skip training of entities.

NLU model training completed.
Your Rasa model is trained and saved at '/content/models/simple.tar.g
z'.
```

```
!rasa test nlu -c config-simple.yml -u test_data.md -m models/simple.
tar.gz --out results/simple --quiet
```

```
report = pd.read_json("results/simple/intent_report.json", orient="va
lues")
```

```
simple_f1 = report["weighted avg"]["f1-
                                   score"]
```

```
data = [["simple", simple_f1]]
```

```
pd.DataFrame(data, columns=["Model", "F1-
                                   sore"])
```

```
!rasa test nlu -c config-simple.yml -u test_data.md -m models/simple.
tar.gz --out results/simple --quiet
```

```
report = pd.read_json("results/simple/intent_report.json", orient="va
lues")
simple_f1 = report["weighted avg"]["f1-score"]
```

```
data = [["simple", simple_f1]]
```

```
pd.DataFrame(data, columns=["Model", "F1-
                                   sore"])
```

```
2020-06-22 21:00:29.891699: E tensorflow/stream_executor/cuda/cuda_dr
iver.cc:351] failed call to cuInit: CUDA_ERROR_NO_DEVICE: no CUDA-cap
able device is detected
100% 14/14 [00:00<00:00, 108.22it/s]
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/metrics/_classificatio
n.py:1272: UndefinedMetricWarning: Precision and F-score are ill-defi
ned and being set to 0.0 in labels with no predicted samples. Use `ze
ro_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.6/dist-packages/sklearn/metrics/_classificatio
n.py:1272: UndefinedMetricWarning: Precision is ill-defined and being
set to 0.0 in labels with no predicted samples. Use `zero_division` p
arameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

```
   Model      F1-sore
0  simple  0.614286
```

Expected F1-score = 0.752381

In test data we have lexically different examples from the ones in training data, so it is expected that our simple pipeline doesn't recognize them properly: `intent:affirm- alright- sure- ok`

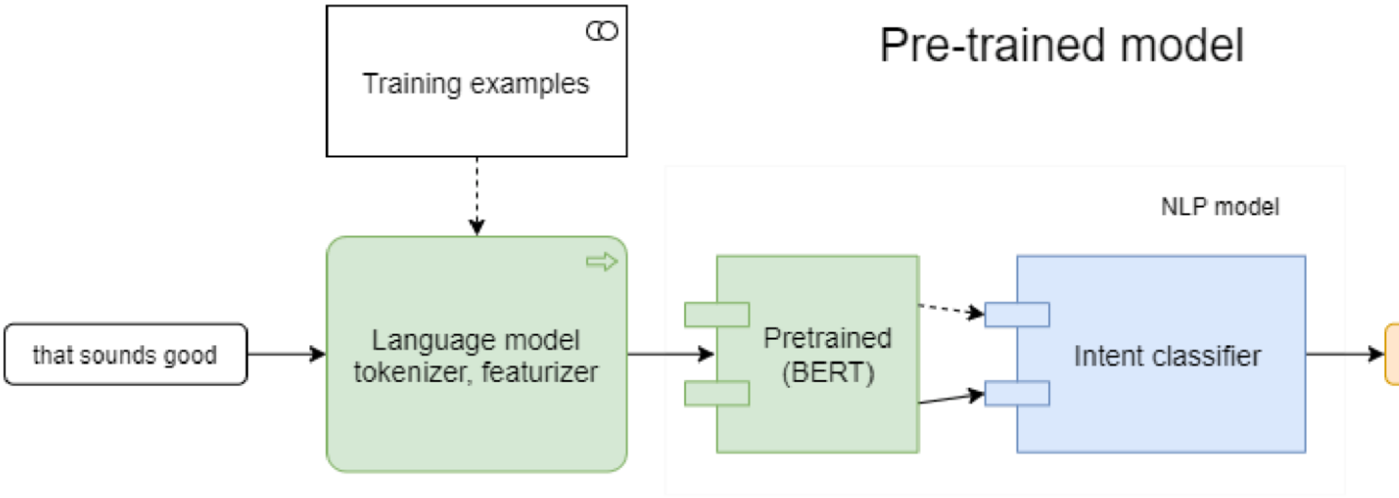
Pre-trained model

The pre-trained language model can be used for NLU tasks without any taskspecific change to the model architecture. Pre-trained models have an ability to continue pre-training on custom data, starting from some checkpoint.

Hire talented developers from LATAM, Canada, and Europe

```
language: en
```

```
pipeline:
```




```
- name: HFTransformersNLP
  model_weights: "bert-base-uncased"
  model_name: "bert"
- name: LanguageModelTokenizer
- name: LanguageModelFeaturizer
```

```
  -             name: DIETClassifier
    epochs: 100
```

```
!rasa train -c config-bert.yml --fixed-model-name bert --quiet
```

```
Core stories/configuration did not change. No need to retrain Core mo
del.
Training NLU model...

Downloading: 100% 232k/232k [00:00<00:00, 1.93MB/s]

Downloading: 100% 433/433 [00:00<00:00, 299kB/s]
Downloading: 100% 536M/536M [00:08<00:00, 63.4MB/s]

2020-06-22 20:48:18.155538: E tensorflow/stream_executor/cuda/cuda_dr
iver.cc:351] failed call to cuInit: CUDA_ERROR_NO_DEVICE: no CUDA-cap
able device is detected
/usr/local/lib/python3.6/dist-packages/rasa/utils/common.py:363: User
Warning: You specified 'DIET' to train entities, but no entities are
present in the training data. Skip training of entities.

NLU model training completed.
```

```
Your Rasa model is trained and saved at '/content/models/bert.tar.gz'
.
```

```
!rasa test nlu -c config-bert.yml -u test_data.md -m models/bert.tar.
gz --out results/bert --quiet
```

```
report = pd.read_json("results/bert/intent_report.json", orient="valu
es")
bert_f1 = report["weighted avg"]["f1-score"]
```

```
data = [["simple", simple_f1], ["bert",
                                bert_f1]]
```

```
pd.DataFrame(data, columns=["Model", "F1-sore"])
```

```
2020-06-22 20:49:03.856455: E tensorflow/stream_executor/cuda/cuda_dr
iver.cc:351] failed call to cuInit: CUDA_ERROR_NO_DEVICE: no CUDA-cap
able device is detected
100% 14/14 [00:03<00:00,  4.04it/s]
```

Model	F1-sore

0	simple	0.614286

1	bert	0.930612

Expected F1-score = 0.930612

As we see, without modification of training data, usage of the pre-trained BERT model improves the accuracy of intent detection. This happens because the model already has knowledge about word's synonyms, which helped to recognize matches.

Fine-tuning your AI chatbot

To perform Fine-tuning of the chatbot development model, follow the instructions on Sentence (and sentence-pair) classification tasks from Google's BERT repository. In general, you need to download some text corpus or to convert your text data to BERT's input format, then run Fine-tuning command. You can prepare a new model with the following script: `from transformers import TFBertModel, BertTokenizer`

```
model = TBertModel.from_pretrained("bert-base-uncased")
model.save_pretrained("./model-fine-tuned-1/")
```

```
tokenizer = BertTokenizer.from_pretrained("bert-base-
uncased")
```

```
tokenizer.save_pretrained("./model-fine-tuned-1/")
```

Follow the text preprocessing steps for fine-tuning. An example

of Finetuning Bert model on the MRPC classification task is given below:

```
export BERT_BASE_DIR=/path/to/bert/uncased_L-12_H-768_A-12

export GLUE_DIR=/path/to/glue



python run_classifier.py \
  --task_name=MRPC \
  --do_train=true \
  --do_eval=true \
  --data_dir=$GLUE_DIR/MRPC \
  --vocab_file=$BERT_BASE_DIR/vocab.txt \
  --bert_config_file=$BERT_BASE_DIR/bert_config.json \
  --init_checkpoint=$BERT_BASE_DIR/bert_model.ckpt \
  --max_seq_length=128 \
  --train_batch_size=32 \
  --learning_rate=2e-5 \
  --num_train_epochs=3.0 \
  --output_dir=/tmp/mrpc_output/
```

When ready, the model from resulting folder can be used in your pipeline and it should have higher F1-score than original one.

Here is another tuning example f

4.Developement part-2

Training Model

Now, we will create the training data in which we will provide the input and the output.

- Our input will be the pattern and output will be the class our input pattern belongs to. But the computer doesn't understand text so we will convert text into numbers

In [9]:

```
# create our training data training = []
# create an empty array for our output output_empty = [0] * len(classes)
# training set, bag of words for each sentence for doc in documents:
   # initialize our bag of words     bag = []
   # list of tokenized words     pattern_words = doc[0]
   # convert pattern_words in lower case
   pattern_words = [lemmatizer.lemmatize(word.lower()) for word in pattern
_words]
   # create bag of words array,if word match found in current pattern then put 1 otherwise 0.[row *
colm(263)]     for w in words:       bag.append(1) if w in pattern_words else bag.append(0)

   # in output array 0 value for each tag ang 1 value for matched tag.[row
* colm(8)]
   output_row = list(output_empty)     output_row[classes.index(doc[1])] = 1

   training.append([bag, output_row]) # shuffle training
and turn into np.array random.shuffle(training) training =
np.array(training)
# create train and test. X - patterns(words), Y - intents(tags) train_x = list(training[:,0])
train_y = list(training[:,1]) print("Training data created")
```

Training data created

In [10]:

linkcode

```
from tensorflow.python.framework import ops ops.reset_default_graph()
```

Build the model

We have our training data ready, now we will build a deep neural network that has 3 layers. We use the Keras sequential API for this. After training the model for 200 epochs, we achieved 100% accuracy on our model. Let us save the model as 'chatbot_model.h5'.

In [11]:

```
# Create model - 3 layers. First layer 128 neurons, second layer 64 neurons and 3rd output layer contains number of neurons
# equal to number of intents to predict output intent with softmax model = Sequential()
```

model.add(Dense(128, input_shape=(len(train_x[0]),), activation='relu')) model.add(Dropout(0.5))
model.add(Dense(64, activation='relu')) model.add(Dropout(0.5))
model.add(Dense(len(train_y[0]), activation='softmax')) print("First layer:",model.layers[0].get_weights()[0])

First layer: [[0.08108504 -0.06599443 -0.10388638 ... -0.01234975 0.0
2568085
0.00633688]
[-0.02540757 -0.0221673 -0.0489299 ... 0.10772091 0.00711305
0.03869867]
[-0.06639696 -0.05009066 -0.03959011 ... -0.0571945 -0.11444904 -0.06228179]
...
[0.02686372 0.0873628 0.12299983 ... -0.07360662 0.05407895
-0.01691054]
[-0.08417445 -0.10581411 -0.07542053 ... -0.06181952 -0.12180413
-0.08388676]
[-0.07259022 0.11421812 -0.04386763 ... 0.00979565 0.05784626 0.09121044]]

In [12]: # *Compile model. Stochastic gradient descent with Nesterov accelerated gradi ent gives good results for this model*
 # sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=[' accuracy'])

In [13]:

#fitting and saving the model
hist = model.fit(np.array(train_x), np.array(train_y), epochs=200, batch_si ze=5, verbose=1)
model.save('chatbot_model.h5', hist)

print("model created") Epoch 1/200

81/81 [==============================] - 1s 2ms/step - loss: 3.6136 - a ccuracy: 0.0543
Epoch 2/200
81/81 [==============================] - 0s 2ms/step - loss: 3.4736 - a ccuracy: 0.1259
Epoch 3/200
81/81 [==============================] - 0s 2ms/step - loss: 3.2848 - a ccuracy: 0.1753
Epoch 4/200
81/81 [==============================] - 0s 2ms/step - loss: 3.0604 - a ccuracy: 0.2346
Epoch 5/200
81/81 [==============================] - 0s 2ms/step - loss: 2.8305 - a ccuracy: 0.2716
Epoch 6/200
81/81 [==============================] - 0s 2ms/step - loss: 2.5375 - a ccuracy: 0.3432
Epoch 7/200
81/81 [==============================] - 0s 2ms/step - loss: 2.3111 - a ccuracy: 0.4025
Epoch 8/200
81/81 [==============================] - 0s 2ms/step - loss: 2.1470 - a ccuracy: 0.4568
Epoch 9/200
81/81 [==============================] - 0s 2ms/step - loss: 1.9539 - a ccuracy: 0.4864
Epoch 10/200
81/81 [==============================] - 0s 2ms/step - loss: 1.7000 - a ccuracy: 0.6025
Epoch 11/200
81/81 [==============================] - 0s 2ms/step - loss: 1.5961 - a ccuracy: 0.6148
Epoch 12/200
81/81 [==============================] - 0s 2ms/step - loss: 1.4055 - a ccuracy: 0.6593
Epoch 13/200
81/81 [==============================] - 0s 2ms/step - loss: 1.3002 - a ccuracy: 0.6963

Epoch 14/200
81/81 [==============================] - 0s 2ms/step - loss: 1.1978 - a ccuracy: 0.6963
Epoch 15/200
81/81 [==============================] - 0s 2ms/step - loss: 1.0640 - a ccuracy: 0.7407
Epoch 16/200
81/81 [==============================] - 0s 2ms/step - loss: 1.0210 - a ccuracy: 0.7506
Epoch 17/200
81/81 [==============================] - 0s 2ms/step - loss: 0.9202 - a ccuracy: 0.7679
Epoch 18/200
81/81 [==============================] - 0s 2ms/step - loss: 0.8287 - a ccuracy: 0.8099
Epoch 19/200
81/81 [==============================] - 0s 2ms/step - loss: 0.7831 - a ccuracy: 0.8198
Epoch 20/200
81/81 [==============================] - 0s 2ms/step - loss: 0.7525 - a ccuracy: 0.8148
Epoch 21/200
81/81 [==============================] - 0s 2ms/step - loss: 0.7355 - a ccuracy: 0.8123 Epoch
22/200
81/81 [==============================] - 0s 2ms/step - loss: 0.6728 - a ccuracy: 0.8272
Epoch 23/200
81/81 [==============================] - 0s 2ms/step - loss: 0.6377 - a ccuracy: 0.8321
Epoch 24/200
81/81 [==============================] - 0s 2ms/step - loss: 0.5440 - a ccuracy: 0.8741
Epoch 25/200
81/81 [==============================] - 0s 2ms/step - loss: 0.4673 - a ccuracy: 0.8889
Epoch 26/200
81/81 [==============================] - 0s 2ms/step - loss: 0.5191 - a ccuracy: 0.8469
Epoch 27/200
81/81 [==============================] - 0s 2ms/step - loss: 0.5168 - a ccuracy: 0.8840
Epoch 28/200
81/81 [==============================] - 0s 2ms/step - loss: 0.4686 - a ccuracy: 0.8864
Epoch 29/200
81/81 [==============================] - 0s 2ms/step - loss: 0.4586 - a ccuracy: 0.8790
Epoch 30/200
81/81 [==============================] - 0s 2ms/step - loss: 0.4126 - a ccuracy: 0.8963
Epoch 31/200
81/81 [==============================] - 0s 2ms/step - loss: 0.4247 - a ccuracy: 0.8889
Epoch 32/200
81/81 [==============================] - 0s 2ms/step - loss: 0.4080 - a ccuracy: 0.8840
Epoch 33/200
81/81 [==============================] - 0s 2ms/step - loss: 0.3659 - a ccuracy: 0.8988
Epoch 34/200
81/81 [==============================] - 0s 2ms/step - loss: 0.4184 - a ccuracy: 0.8889
Epoch 35/200
81/81 [==============================] - 0s 2ms/step - loss: 0.3590 - a ccuracy: 0.9062
Epoch 36/200
81/81 [==============================] - 0s 2ms/step - loss: 0.3597 - a ccuracy: 0.9185
Epoch 37/200
81/81 [==============================] - 0s 2ms/step - loss: 0.3258 - a ccuracy: 0.9111
Epoch 38/200
81/81 [==============================] - 0s 2ms/step - loss: 0.3448 - a ccuracy: 0.9111
Epoch 39/200
81/81 [==============================] - 0s 2ms/step - loss: 0.2794 - a ccuracy: 0.9259
Epoch 40/200

81/81 [==============================] - 0s 2ms/step - loss: 0.3334 - a ccuracy: 0.9012
Epoch 41/200
81/81 [==============================] - 0s 2ms/step - loss: 0.3310 - a ccuracy: 0.9037
Epoch 42/200
81/81 [==============================] - 0s 2ms/step - loss: 0.2302 - a ccuracy: 0.9407
Epoch 43/200
81/81 [==============================] - 0s 2ms/step - loss: 0.2965 - a ccuracy: 0.9185
Epoch 44/200
81/81 [==============================] - 0s 2ms/step - loss: 0.2444 - a ccuracy: 0.9333
Epoch 45/200
81/81 [==============================] - 0s 2ms/step - loss: 0.2701 - a ccuracy: 0.9210
Epoch 46/200
81/81 [==============================] - 0s 2ms/step - loss: 0.3027 - a ccuracy: 0.9309
Epoch 47/200
81/81 [==============================] - 0s 3ms/step - loss: 0.2240 - a ccuracy: 0.9531
Epoch 48/200
81/81 [==============================] - 0s 2ms/step - loss: 0.2129 - a ccuracy: 0.9432
Epoch 49/200
81/81 [==============================] - 0s 2ms/step - loss: 0.2348 - a ccuracy: 0.9407
Epoch 50/200
81/81 [==============================] - 0s 2ms/step - loss: 0.2572 - a ccuracy: 0.9358
Epoch 51/200
81/81 [==============================] - 0s 2ms/step - loss: 0.2377 - a ccuracy: 0.9259
Epoch 52/200
81/81 [==============================] - 0s 2ms/step - loss: 0.2324 - a ccuracy: 0.9358
Epoch 53/200
81/81 [==============================] - 0s 2ms/step - loss: 0.2190 - a ccuracy: 0.9407
Epoch 54/200
81/81 [==============================] - 0s 2ms/step - loss: 0.2175 - a ccuracy: 0.9432 Epoch
55/200
81/81 [==============================] - 0s 2ms/step - loss: 0.2259 - a ccuracy: 0.9160
Epoch 56/200
81/81 [==============================] - 0s 2ms/step - loss: 0.2127 - a ccuracy: 0.9481
Epoch 57/200
81/81 [==============================] - 0s 2ms/step - loss: 0.1997 - a ccuracy: 0.9457
Epoch 58/200
81/81 [==============================] - 0s 2ms/step - loss: 0.1975 - a ccuracy: 0.9407
Epoch 59/200
81/81 [==============================] - 0s 2ms/step - loss: 0.2083 - a ccuracy: 0.9333
Epoch 60/200
81/81 [==============================] - 0s 2ms/step - loss: 0.2078 - a ccuracy: 0.9407
Epoch 61/200
81/81 [==============================] - 0s 2ms/step - loss: 0.1838 - a ccuracy: 0.9432
Epoch 62/200
81/81 [==============================] - 0s 2ms/step - loss: 0.1736 - a ccuracy: 0.9506
Epoch 63/200
81/81 [==============================] - 0s 2ms/step - loss: 0.2022 - a ccuracy: 0.9407
Epoch 64/200
81/81 [==============================] - 0s 2ms/step - loss: 0.1883 - a ccuracy: 0.9481

Evaluation:

Intelligent ChatBot built with Microsoft's DialoGPT transformer to make conversations with human users!

Image by Andy Kelly

What is a chatbot?

A ChatBot is a kind of virtual assistant that can build conversations with human users! A Chatting Robot. Building a chatbot is one of the popular tasks in Natural Language Processing.

Are all chatbots the same?

Chatbots fall under three common categories:

1. Rule-based chatbots

2. Retrieval-based chatbots

3. Intelligent chatbots

Rule-based chatbots

These bots respond to users' inputs based on certain pre-specified rules. For instance, these rules can be defined as if-elif-else statements. While writing rules for these chatbots, it is important to expect all possible user inputs, else the bot may fail to answer properly. Hence, rule-based chatbots do not possess any cognitive skills.

Retrieval-based chatbots

These bots respond to users' inputs by retrieving the most relevant information from the given text document. The most relevant information can be determined by Natural Language Processing with a scoring system such as cosine-similarity-score. Though

these bots use NLP to do conversations, they lack cognitive skills to match a real human chatting companion.

Intelligent AI chatbots

These bots respond to users' inputs after understanding the inputs, as humans do. These bots are trained with a Machine Learning Model on a large training dataset of human conversations. These bots are cognitive to match a human in conversing. Amazon's Alexa, Apple's Siri fall under this category. Further, most of these bots can make conversations based on the preceding chat texts.

In this Article?

This article describes building an intelligent AI chatbot based on the famous transformer architecture - Microsoft's DialoGPT. According to Hugging Face's model card, DialoGPT is a State-Of-The-Art large-scale pretrained dialogue response generation model for multiturn conversations. The human evaluation results indicate that the response generated from DialoGPT is comparable to human response quality under a single-turn conversation Turing test. The model is trained on 147M multi-turn dialogue from Reddit discussion thread.

Let's Python

Import necessary libraries and frameworks

In [1]:

```
import numpy as np
import time import os
from transformers import AutoModelForCausalLM, AutoTokenizer import torch
```

Download Microsoft's DialoGPT model and tokenizer

The Hugging Face checkpoint for the model and its tokenizer is "microsoft/DialoGPTmedium"

In [2]:

```
# checkpoint
checkpoint = "microsoft/DialoGPT-medium"
# download and cache tokenizer
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
# download and cache pre-trained model
model = AutoModelForCausalLM.from_pretrained(checkpoint)
```

A ChatBot class

In [3]:

linkcode

```
# Build a ChatBot class with all necessary modules to make a complete conver sation class ChatBot():     #
initialize     def __init__(self):
        # once chat starts, the history will be stored for chat continuity        self.chat_history_ids = None
        # make input ids global to use them anywhere within the object        self.bot_input_ids = None
        # a flag to check whether to end the conversation        self.end_chat = False
# greet while starting        self.welcome()
            def welcome(self):        print("Initializing ChatBot
...")        # some time to get user ready        time.sleep(2)
        print('Type "bye" or "quit" or "exit" to end chat \n')
```

```
        # give time to read what has been printed         time.sleep(3)
# Greet and introduce         greeting = np.random.choice([
            "Welcome, I am ChatBot, here for your kind service",
            "Hey, Great day! I am your virtual assistant",
            "Hello, it's my pleasure meeting you",
            "Hi, I am a ChatBot. Let's chat!"
        ])
        print("ChatBot >>  " + greeting)              def
user_input(self):         # receive input from user         text =
input("User    >> ")         # end conversation if user wishes
so         if text.lower().strip() in ['bye', 'quit', 'exit']:
            # turn flag on
self.end_chat=True             # a closing
comment
            print('ChatBot >>  See you soon! Bye!')            time.sleep(1)
            print('\nQuitting ChatBot ...')         else:
            # continue chat, preprocess input text
            # encode the new user input, add the eos_token and return a tens or in Pytorch
            self.new_user_input_ids = tokenizer.encode(text + tokenizer.eos
_token, \
                                                       return_tensors='pt')
     def bot_response(self):
        # append the new user input tokens to the chat history
        # if chat has already begun         if self.chat_history_ids is not None:            self.bot_input_ids =
torch.cat([self.chat_history_ids, self.new
_user_input_ids], dim=-1)          else:
            # if first entry, initialize bot_input_ids            self.bot_input_ids = self.new_user_input_ids

        # define the new chat_history_ids based on the preceding chats         # generated a response while
limiting the total chat history to 1000 tokens,
        self.chat_history_ids = model.generate(self.bot_input_ids, max_leng th=1000, \
                                               pad_token_id=tokenizer.eos_t oken_id)
        # last ouput tokens from bot
        response = tokenizer.decode(self.chat_history_ids[:, self.bot_input
_ids.shape[-1]:][0], \
                               skip_special_tokens=True)
        # in case, bot fails to answer         if response == "":
response = self.random_response()
        # print bot response
        print('ChatBot >>  '+ response)

    # in case there is no response from model     def
random_response(self):
        i = -1
        response = tokenizer.decode(self.chat_history_ids[:, self.bot_input _ids.shape[i]:][0], \
                               skip_special_tokens=True)         # iterate over history backwards to find
the last token         while response == '':            i = i-1
            response = tokenizer.decode(self.chat_history_ids[:, self.bot_i nput_ids.shape[i]:][0], \
                               skip_special_tokens=True)
        # if it is a question, answer suitably          if response.strip() == '?':
reply = np.random.choice(["I don't know",                                "I am not
sure"])
        # not a question? answer suitably         else:            reply =
np.random.choice(["Great",
```

"Fine. What's up?",

CONCLUSION:

Creating a chatbot in Python can be a rewarding and versatile project, but it comes with its own set of challenges and considerations. In conclusion, here are some key takeaways:

1. **Versatility**: Python is an excellent choice for building chatbots due to its vast libraries and frameworks. You can create chatbots for various platforms, such as web, desktop, or messaging apps.

2. **Natural Language Processing (NLP)**: Successful chatbots rely on NLP libraries like NLTK, spaCy, or TensorFlow to understand and generate human-like responses. These libraries help the chatbot comprehend user input and respond appropriately.

3. **Dialog Management**: Effective chatbots must manage conversations coherently, remembering context, and handling interruptions. Building a robust dialog management system is crucial for a seamless user experience.

4. **User Experience**: Ensuring a positive user experience is paramount. The chatbot's responses should be clear, concise, and relevant. Testing with real users and iterating on feedback is essential for improvement.

5. **Data Collection and Training**: Data is key. You'll need a substantial dataset to train your chatbot, and it should be continuously updated to stay relevant. You may also need to fine-tune the model for specific use cases.

6. **Integration**: Depending on your chatbot's purpose, you might need to integrate it with external services and APIs. Python's extensive library support makes this process relatively straightforward.

7. **Security and Privacy**: Be mindful of user data and privacy concerns. Implement secure data handling practices and ensure that the chatbot doesn't inadvertently leak sensitive information.

8. **Scalability**: As your chatbot gains users, you need to ensure it can scale to handle increased traffic. Consider deploying it on cloud platforms for scalability.

9. **Maintenance**: Chatbots are not "set and forget" projects. Regular maintenance is essential to keep them up-to-date, fix issues, and improve their conversational abilities.

10. **Testing and Quality Assurance**: Extensive testing is vital to catch and correct any issues. You should have a robust testing strategy, including automated tests and real user testing.

11. **Legal and Ethical Considerations**: Be aware of legal and ethical considerations, especially if your chatbot interacts with users in sensitive domains. Compliance with regulations like GDPR is crucial.