CREATE A CHATBOT USING PYTHON

PHASE 4 : DEVELOPMENT PART 2

SUBMISSION DOCUMENT.....

Project : Create Chatbot Using Python.

Introduction:

- Creating a chatbot using Python involves developing a program that can simulate human conversation.
- You'll need to decide on the type of chatbot, such as rule-based or AI-powered, select the right tools and libraries, design the conversation flow, write the code, test it thoroughly, and then deploy it.
- Continuous maintenance and improvement are crucial for keeping your chatbot effective.
- Python's versatility and the availability of NLP and ML libraries make it a powerful choice for chatbot development, offering the potential to enhance user experiences, automate tasks, and provide valuable support in various applications.

Content for phase 4 in project:

In this sphase 4 , we start to build the project by performing different activities like feature engineering, model training, evaluation etc as per the instructions in the project.

Data Source :

A good data source for creating a chatbot should contain accurate ,complete ,and easy accessible one for users.

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

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.
it's okay. it's a really big campus. good luck with school.
good luck with school. thank you very much.
how's it going? i'm doing well. how about you?
i'm doing well. how about you? never better, thanks.
never better, thanks. so how have you been lately?
so how have you been lately? i've actually been pretty good. you?
i've actually been pretty good. you? i'm actually in school right now.
i'm actually in school right now. which school do you attend?
which school do you attend? i'm attending pcc right now.
i'm attending pcc right now. are you enjoying it there?
are you enjoying it there? it's not bad. there are a lot of people there.
it's not bad. there are a lot of people there. good luck with that.
good luck with that. thanks.
how are you doing today? i'm doing great. what about you?
i'm doing great. what about you? i'm absolutely lovely, thank you.
i'm absolutely lovely, thank you. everything's been good with you?
everything's been good with you? i haven't been better. how about yourself?
i haven't been better. how about yourself? i started school recently.
i started school recently. where are you going to school?

where are you going to school? i'm going to pe.....

.....

Feature Engineering:

Feature engineering is the process of selecting, transforming, and creating relevant input data (features) to improve the performance of machine learning models, such as those used in chatbots. In the context of chatbots, feature engineering may include:

1) Text Representation: Converting text data into numerical vectors, often using techniques like TF-IDF (Term Frequency-Inverse Document Frequency) or word embeddings (e.g., Word2Vec or GloVe).
2) Contextual Features: Capturing context and conversation history to provide meaningful context to the chatbot's responses.
3) Entity Recognition: Identifying entities (e.g., names, dates, locations) in the text, which can be useful for responding to user queries.
4) Sentiment Analysis: Determining the sentiment or emotional tone of the conversation, which can help tailor responses.
5) Intent Recognition: Identifying the user's intent to route the conversation appropriately.
6) Language Processing: Preprocessing and cleaning text data, which includes tasks like tokenization and removing stopwords.

Model Training:

Model training involves the process of building and training a machine learning or natural language processing model to enable the chatbot to understand and generate text-based responses. Common model types for chatbots include:

1) Rule-Based Models: These models rely on predefined rules to generate responses based on specific patterns or keywords in user input.
2) Retrieval-Based Models: Retrieval-based models select predefined responses from a set of responses based on the input. They often use methods like TF-IDF or similarity scoring.
3) Generative Models: Generative models, like sequence-to-sequence models, can generate responses from scratch, making them capable of more creative and context-aware responses.

Training a model typically involves using a labeled dataset of conversations or user interactions to learn patterns and develop the ability to generate relevant responses.

Evaluation:

Evaluation is a crucial step to assess the performance and effectiveness of a chatbot. It involves measuring how well the chatbot responds to user input. Common evaluation metrics for chatbots include:

1) Accuracy: Measures how often the chatbot provides a correct response.
2) F1 Score: Combines precision and recall to evaluate the model's ability to correctly identify and respond to different intents or questions.
3) BLEU Score: Often used for generative models, this metric evaluates the quality of generated text by comparing it to reference text.
4) User Satisfaction: Surveys or user feedback to gauge user satisfaction with the chatbot's responses.

Continuous evaluation is essential to identify areas for improvement and ensure the chatbot is providing valuable interactions to users.

In the process of creating a chatbot, these concepts work together to build a system that can understand and respond to user queries effectively. The choice of feature engineering techniques, model type, and evaluation metrics will depend on the specific goals and requirements of the chatbot project.

Model :

Bahdanau Attention Mechanism

https://machinelearningmastery.com/the-bahdanau-attention-mechanism/

Adding attention mechanism to an Encoder-Decoder Model to make the model focus on sp ecific parts of input sequence by assigning weights to different parts of the input sequence

Buliding Model Architecture :

Adding attention mechanism to an Encoder-Decoder Model to make the m odel focus on specific parts of input sequence by assigning weights to different parts of the input sequence

Buliding Model Architecture,

Encoder :

```
class Encoder(tf.keras.Model):
  def __init__(self, vocab_size, embedding_dim, enc_units, batch_sz):
    super(Encoder, self).__init__()
    self.batch_sz = batch_sz
```

```
        self.enc_units = enc_units
        self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
        self.gru = tf.keras.layers.GRU(self.enc_units,
                                       return_sequences=True,
                                       return_state=True,
                                       recurrent_initializer='glorot_uniform')

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

    def initialize_hidden_state(self):
        return tf.zeros((self.batch_sz, self.enc_units))
encoder = Encoder(vocab_inp_size, embedding_dim, units, BATCH_SIZE)

# sample input

sample_hidden = encoder.initialize_hidden_state()
sample_output, sample_hidden = encoder(example_input_batch, sample_hidden)
print ('Encoder output shape: (batch size, sequence length, units) {}'.format(sample_output.
shape))
print ('Encoder Hidden state shape: (batch size, units) {}'.format(sample_hidden.shape))
Encoder output shape: (batch size, sequence length, units) (64, 24, 1024)
Encoder Hidden state shape: (batch size, units) (64, 1024)

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

    def call(self, query, values):
        # query hidden state shape == (batch_size, hidden size)
        # query_with_time_axis shape == (batch_size, 1, hidden size)
        # values shape == (batch_size, max_len, hidden size)
        # we are doing this to broadcast addition along the time axis to calculate the score
        query_with_time_axis = tf.expand_dims(query, 1)

        # score shape == (batch_size, max_length, 1)
        # we get 1 at the last axis because we are applying score to self.V
        # the shape of the tensor before applying self.V is (batch_size, max_length, units)
```

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

        # attention_weights shape == (batch_size, max_length, 1)
        attention_weights = tf.nn.softmax(score, axis=1)

        # context_vector shape after sum == (batch_size, hidden_size)
        context_vector = attention_weights * values
        context_vector = tf.reduce_sum(context_vector, axis=1)

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

print("Attention result shape: (batch size, units) {}".format(attention_result.shape))
print("Attention weights shape: (batch_size, sequence_length, 1) {}".format(attention_weig
hts.shape))
Attention result shape: (batch size, units) (64, 1024)
Attention weights shape: (batch_size, sequence_length, 1) (64, 24, 1)
```

Decoder :

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

        # used for attention
        self.attention = BahdanauAttention(self.dec_units)

    def call(self, x, hidden, enc_output):
        # enc_output shape == (batch_size, max_length, hidden_size)
        context_vector, attention_weights = self.attention(hidden, enc_output)

        # x shape after passing through embedding == (batch_size, 1, embedding_dim)
        x = self.embedding(x)
```

```
        # x shape after concatenation == (batch_size, 1, embedding_dim + hidden_size)
        x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1)

        # passing the concatenated vector to the GRU
        output, state = self.gru(x)

        # output shape == (batch_size * 1, hidden_size)
        output = tf.reshape(output, (-1, output.shape[2]))

        # output shape == (batch_size, vocab)
        x = self.fc(output)

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

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

print ('Decoder output shape: (batch_size, vocab size) {}'.format(sample_decoder_output.s
hape))
Decoder output shape: (batch_size, vocab size) (64, 2349)
```

Training Model :

1) Pass the input through the encoder which return encoder output and the encoder hid den state.
2) The encoder output, encoder hidden state and the decoder input (which is the start t oken) is passed to the decoder.
3) The decoder returns the predictions and the decoder hidden state.
4) The decoder hidden state is then passed back into the model and the predictions are used to calculate the loss.
5) Use teacher forcing to decide the next input to the decoder.
6) Teacher forcing is the technique where the target word is passed as the next input to the decoder.
7) The final step is to calculate the gradients and apply it to the optimizer and backprop agate.

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

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

```
    mask = tf.cast(mask, dtype=loss_.dtype)
    loss_ *= mask

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

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

        dec_hidden = enc_hidden

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

        # Teacher forcing - feeding the target as the next input
        for t in range(1, targ.shape[1]):
            # passing enc_output to the decoder
            predictions, dec_hidden, _ = decoder(dec_input, dec_hidden, enc_output)

            loss += loss_function(targ[:, t], predictions)

            # using teacher forcing
            dec_input = tf.expand_dims(targ[:, t], 1)

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

    variables = encoder.trainable_variables + decoder.trainable_variables

    gradients = tape.gradient(loss, variables)

    optimizer.apply_gradients(zip(gradients, variables))

    return batch_loss
EPOCHS = 40

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

    for (batch, (inp, targ)) in enumerate(dataset.take(steps_per_epoch)):
        batch_loss = train_step(inp, targ, enc_hidden)
        total_loss += batch_loss

    if(epoch % 4 == 0):
```

```
    print('Epoch:{:3d} Loss:{:.4f}'.format(epoch,
                                  total_loss / steps_per_epoch))
```

Model Evaluation :

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

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

  result = ''

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

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

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

    # storing the attention weights to plot later on
    attention_weights = tf.reshape(attention_weights, (-1, ))

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

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

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

    # the predicted ID is fed back into the model
    dec_input = tf.expand_dims([predicted_id], 0)
```

```
    return remove_tags(result), remove_tags(sentence)
def ask(sentence):
    result, sentence = evaluate(sentence)

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

CONCLUSION AND FUTURE WORK(Phase 3) :

Project Conclusion :

- In the Phase 4 conclusion, building a chatbot is a multifaceted and iterative process. It involves careful consideration of feature engineering to provide context and understanding, selecting and training a suitable model to generate responses, and evaluating the chatbot's performance to ensure it meets the desired objectives. The success of the chatbot project depends on a systematic approach to these activities, along with a commitment to ongoing improvement and refinement.
- Future Work: we'll document and complete the chatbot project fully, incorporating feature engineering, model training, and evaluation to create an effective conversational agent.