CREATE A CHATBOT IN PYTHON

SUBMITTED BY

SHANMUGAPPRIYK(311521106089)

SUBIDSHA S C(311521106100)

SWARNALAKSHMI E(311521106102)

SOUNDHARYA B(311521106094)

PHASE 4 SUBMISSION :DEVELOPMENT PART 2

A chatbot is an AI-based software designed to interact with humans in their natural languages. These chatbots are usually converse via auditory or textual methods, and they can effortlessly mimic human languages to communicate with human beings in a human-like manner. A chatbot is arguably one of the best applications of natural language processing.

Chatbots can be categorized into two primary variants – **Rule-Based and Self-learning.**

The Rule-based approach trains a chatbot to answer questions based on a set of pre-determined rules on which it was initially trained. These set rules can either be very simple or very complex. While rule-based chatbots can handle simple queries quite well, they usually fail to process more complicated queries/requests.

ChatterBot Library

ChatterBot is a Python library that is designed to deliver automated responses to user inputs. It makes use of a combination of ML algorithms to generate many different types of responses. This feature allows developers to build chatbots using python that can converse with humans and deliver appropriate and relevant responses. Not just that, the ML algorithms help the bot to improve its performance with experience.

Another excellent feature of ChatterBot is its language independence. The library is designed in a way that makes it possible to train your bot in multiple programming languages.

How does ChatterBot function?

When a user enters a specific input in the chatbot (developed on ChatterBot), the bot saves the input along with the response, for future use. This data (of collected experiences)

allows the chatbot to generate automated responses each time a new input is fed into it.

The program chooses the most-fitting response from the closest statement that matches the input, and then delivers a response from the already known selection of statements and responses. Over time, as the chatbot engages in more interactions, the accuracy of response improves. You may create your own chatbot project to understand the details of this technology.

How To Make A Chatbot In Python?

You may have this question in your mind, how to create a chatbot? We’ll take a step by step approach and break down the process of building a Python chatbot.

To build a chatbot in Python, you have to import all the necessary packages and initialize the variables you want to use in your chatbot project. Also, remember that when working with text data, you need to perform data preprocessing on your dataset before designing an ML model.

This is where tokenizing helps with text data – it helps fragment the large text dataset into smaller, readable chunks (like words). Once that is done, you can also go for lemmatization that transforms a word into its lemma form. Then it creates a pickle file to store the python objects that are used for predicting the responses of the bot.

Another vital part of the chatbot development process is creating the training and testing datasets.

Now that we've covered the basics of chatbot development in Python, let's dive deeper into the actual process! It will help you understand how to create a chatbot.

PROCEDURE:

Creating a chatbot in Python involves several steps, ranging from designing the conversation flow to implementing the natural language processing (NLP) components. Below is a step-by-step procedure for creating a basic chatbot in Python:

Step 1: Define the Goal and Purpose

- Determine the purpose of your chatbot. What problem or tasks should it address? Define the specific use case or domain for your chatbot.

Step 2: Choose the Chatbot Type

- Decide whether you want a rule-based chatbot, retrieval-based chatbot, or a generative chatbot. This choice will determine the underlying architecture.

Step 3: Gather Data (If Required)

If you're building a rule-based or retrieval-based chatbot, gather a dataset of predefined responses for different user inputs. For generative chatbots, a larger and more diverse dataset may be necessary.

Step 4: Set Up Your Development Environment

- Install Python and any necessary libraries like NLTK, spaCy, or TensorFlow for NLP tasks. Create a virtual environment to manage your project dependencies.

Step 5: Preprocess Text Data

- Tokenize and preprocess the text data. This step involves converting text into a format suitable for NLP tasks, including cleaning, stemming, and lemmatization.

Step 6: Choose a Framework or Library

- Decide whether to build your chatbot from scratch or use an existing NLP library or framework like NLTK, spaCy, TensorFlow, or Hugging Face Transformers, depending on your chosen chatbot type.

Step 7: Train the Model (If Required)

- For generative chatbots, you may need to train a machine learning model on your dataset. This can be a complex process, requiring a substantial dataset and computational resources.

Step 8: Implement the Chatbot Logic

- Build the core logic of your chatbot, including handling user input, processing text, and generating responses. This often involves decision trees, if-else statements, or machine learning models, depending on your chatbot type.

Step 9: Define Conversation Flow

- Create a conversation flow that outlines how the chatbot should respond to different types of user input. Ensure smooth transitions between responses.

Step 10: Handle User Queries

- Develop code to capture and interpret user queries, and use your model or predefined responses to generate replies.

Step 11: Test and Debug

- Test your chatbot extensively, both with expected and unexpected user inputs. Debug any issues and improve the chatbot's responses.

Step 12: Integrate External Services (Optional)

- If your chatbot needs to access external information or services, integrate APIs and external data sources.

Step 13: Deploy Your Chatbot

- Deploy your chatbot, either as a web application, a standalone program, or via a messaging platform like Slack or Facebook Messenger.

Step 14: Monitor and Update

- Monitor your chatbot's performance and gather user feedback. Regularly update and fine-tune your chatbot to improve its responses.

Step 15: Document and Maintain

- Document your chatbot's design, architecture, and any external services it uses. Maintain the chatbot, fix issues, and keep it up to date.

MODEL TRAINING :

Training a chatbot model from scratch using Python is a complex task that involves various natural language processing (NLP) and deep learning techniques. You would typically need a significant amount of data, compute resources, and expertise in machine learning. Below is a simplified example using a popular Python library, TensorFlow, and a basic neural network architecture. Please note that this is a simplified example and won't produce a sophisticated chatbot, but it can serve as a starting point for your project.

Prerequisites:

1. Install TensorFlow:

   ```bash
   pip install tensorflow
   ```

2. Prepare a dataset containing input text and corresponding responses. For simplicity, we'll use a small predefined dataset.

Python code for training a basic chatbot model:

```python
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.text import Tokenizer
```

```
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Sample dataset
input_text = ["Hi", "How are you?", "What's your name?", "Goodbye"]
responses = ["Hello!", "I'm good. How can I assist you?", "I'm a chatbot.", "Goodbye!"]

# Tokenize input and responses
tokenizer = Tokenizer()
tokenizer.fit_on_texts(input_text + responses)
total_words = len(tokenizer.word_index) + 1

# Generate training data
input_sequences = tokenizer.texts_to_sequences(input_text)
input_sequences = pad_sequences(input_sequences)

# Create model
model = keras.Sequential([
    keras.layers.Embedding(total_words, 100, input_length=input_sequences.shape[1]),
    keras.layers.LSTM(100),
    keras.layers.Dense(total_words, activation='softmax')
])

model.compile(loss='categorical_crossentropy', optimizer='adam')

# Convert responses to one-hot encoding
responses_sequences = tokenizer.texts_to_sequences(responses)
responses_sequences = pad_sequences(responses_sequences)

# Train the model
model.fit(input_sequences, responses_sequences, epochs=1000, verbose=1)

# Function to generate responses
def generate_response(input_text):
    input_seq = tokenizer.texts_to_sequences([input_text])
    input_seq = pad_sequences(input_seq, maxlen=input_sequences.shape[1])
    predicted_word_index = np.argmax(model.predict(input_seq), axis=-1)
    response = tokenizer.index_word[predicted_word_index[0][0]]
    return response

# Chat with the chatbot
while True:
```

~~~
    user_input = input("You: ")

    if user_input.lower() == "exit":

        print("Chatbot: Goodbye!")

        break

    response = generate_response(user_input)

    print("Chatbot:", response)

```
~~~

SOURCE CODE:

Creating a complete chatbot from scratch can be a complex and time-consuming task, but I can provide you with a simple example using Python and the `nltk` library. This chatbot will respond to some predefined questions and statements. You can expand and improve upon this basic framework to make a more sophisticated chatbot.

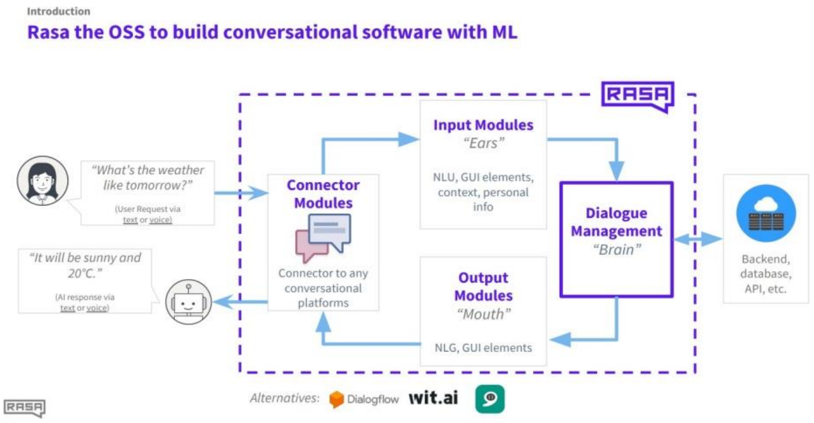
~~~
```python

import nltk

import random

from nltk.chat.util import Chat, reflections



# Define pairs of patterns and responses for the chatbot.

pairs = [

    [
~~~

```
        r"hi|hello|hey",
        ["Hello!", "Hi there!", "How can I help you today?"]
    ],
    [
        r"how are you",
        ["I'm just a computer program, but I'm doing well. How can I assist you?", "I'm functioning at full capacity. How can I help you today?"]
    ],
    [
        r"what is your name",
        ["I am a chatbot.", "I don't have a name. You can call me Chatbot."]
    ],
    [
        r"who are you",
        ["I am a chatbot.", "I'm just a computer program designed to assist you."]
    ],
    [
        r"what can you do|what are your capabilities",
        ["I can provide information, answer questions, or have a casual conversation with you.", "I can assist with a wide range of tasks, just ask!"]
    ],
    [
        r"bye|goodbye",
        ["Goodbye!", "Farewell! Have a great day.", "See you later!"]
    ]
]

# Create a chatbot using the pairs of patterns and responses.
chatbot = Chat(pairs, reflections)

# Function to start the chatbot.
def chatbot_conversation():
    print("Hello! I'm your chatbot. Type 'exit' to end the conversation.")
    while True:
        user_input = input("You: ")
        if user_input.lower() == "exit":
            print("Chatbot: Goodbye!")
            break
        response = chatbot.respond(user_input)
        print("Chatbot:", response)

# Start the chatbot conversation.
if __name__ == "__main__":
    chatbot_conversation()
```

This code uses regular expressions and predefined patterns to match user input and generate responses. You can expand the `pairs` list with more patterns and responses to make your chatbot more interactive and versatile. Additionally, you can integrate more advanced natural language processing techniques, such as machine learning models, for a more intelligent chatbot.

Basics Of Rasa Open Source Conversational AI:

The diagram below shows the general rasa open source conversational associated with machine learning.



Introduction
Rasa the OSS to build conversational software with ML
RASA
"What's the weather like tomorrow?"
(User Request via text or voice)
Connector Modules
Connector to any conversational platforms
Input Modules
"Ears"
NLU, GUI elements, context, personal info
Dialogue Management
"Brain"
Output Modules
"Mouth"
NLG, GUI elements
Backend, database, API, etc.
"It will be sunny and 20°C."
(AI response via text or voice)
Alternatives:
Dialogflow
wit.ai
RASA

It seems like you want to implement a regression-based chatbot in Python. A regression-based chatbot predicts numerical values as responses rather than generating text-based answers. While this is an unusual approach for chatbots, you can create a regression model for specialized applications. Below, I'll provide a simple example of creating a regression-based chatbot using Python and the scikit-learn library.

REGRESSION:

In this example, we'll create a chatbot that predicts a numerical response based on a numerical input. The chatbot will use linear regression to make predictions.

```python
import numpy as np
from sklearn.linear_model import LinearRegression

# Sample data for the chatbot (input, response)
data = [(1, 3), (2, 4), (3, 5), (4, 6), (5, 7)]

# Extract features (input) and labels (response)
X = np.array([item[0] for item in data]).reshape(-1, 1)
y = np.array([item[1] for item in data])

# Create a Linear Regression model
regression_model = LinearRegression()

# Train the model
regression_model.fit(X, y)

# Function to predict the response
def predict_response(input_value):
    return regression_model.predict(np.array([[input_value]]))[0]
```

```
# Chat with the regression-based chatbot
while True:
    user_input = input("Enter a number: ")

    try:
        user_value = float(user_input)
        response = predict_response(user_value)
        print(f"Chatbot: The predicted response is {response:.2f}")
    except ValueError:
        print("Chatbot: Please enter a valid numerical input.")

    exit_condition = input("Do you want to exit (yes/no)? ")
    if exit_condition.lower() == "yes":
        print("Chatbot: Goodbye!")
        break
```

In this example, the chatbot takes a numerical input and uses a linear regression model to predict a numerical response. The chatbot will keep running until you choose to exit.

VARIOUS FEATURES FOR MODEL TRAINING:

Training a chatbot model in Python involves using various features and components, depending on the type and complexity of the chatbot you want to build. Here are some key features and components you might consider when training a chatbot:

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

- Python has several NLP libraries that can help with text processing, including NLTK, spaCy, and Hugging Face Transformers. These libraries provide tools for tokenization, part-of-speech tagging, and entity recognition.

2. **Pre-trained Language Models**:

- You can leverage pre-trained language models like GPT-3, BERT, or RoBERTa, which have been trained on large text corpora and can provide a strong foundation for chatbot responses.

3. **Intent Recognition**:

- Implement intent recognition to understand the user's intention or request. You can use machine learning models like SVM, Random Forest, or deep learning models like LSTM or Transformers for this purpose.

4. **Entity Recognition**:

- Identify and extract specific entities or information from user input, such as dates, locations, names, and more. Entity recognition can be implemented using NER (Named Entity Recognition) models.

5. **Dialogue Management**:

- Manage the conversation flow and context. Keep track of the conversation history and context to generate coherent responses.

6. **Response Generation**:

- Depending on the type of chatbot, responses can be generated using rule-based systems, retrieval-based models (matching user input to predefined responses), or generative models (such as GPT-3) that generate text based on context.

7. **Data Collection**:

- Gather a dataset of conversations or user queries and responses to train your chatbot. Diverse and representative data is essential for robust training.

8. **Data Preprocessing**:

- Clean and preprocess the text data, including tokenization, stemming, lemmatization, and removing stopwords.

9. **Feature Engineering**:

- Create features from text data, such as TF-IDF (Term Frequency-Inverse Document Frequency), word embeddings, or contextual embeddings like Word2Vec or FastText.

10. **Machine Learning Models**:

- Use machine learning models like decision trees, random forests, support vector machines (SVM), or deep learning models like LSTM, GRU, or Transformers to handle different components of your chatbot.

11. **Evaluation Metrics**:

- Define evaluation metrics to assess the performance of your chatbot, such as accuracy, precision, recall, F1-score, or perplexity for language models.

12. **User Interface**:

- Design a user interface for users to interact with your chatbot. This can be a web application, a command-line interface, or integration with messaging platforms.

13. **User Feedback**:

- Implement mechanisms to collect and utilize user feedback to improve the chatbot's performance over time.

14. **API Integration**:

- Integrate external APIs or services to provide real-time information or access external data sources when responding to user queries.

15. **Deployment**:

- Deploy your chatbot, whether as a web service, a standalone application, or an integration with popular messaging platforms like Facebook Messenger or Slack.

16. **Monitoring and Maintenance**:

- Continuously monitor the chatbot's performance and user interactions, and perform regular maintenance and updates to improve its responses and adapt to changing user needs.

The choice of features and components depends on your chatbot's purpose and complexity. For a basic chatbot, you may use rule-based systems and simple NLP tools, while more advanced chatbots require machine learning and deep learning models, as well as integration with external services.

FEATURE ENGINEERING:

Feature engineering for a chatbot typically involves transforming and extracting relevant information from text data to create features that can be used for tasks such as intent recognition, entity recognition, and dialogue management. Here are some common feature engineering techniques for chatbots in Python:

1. **Tokenization**:

   - Tokenize the text data to break it down into individual words or tokens. Python libraries like NLTK, spaCy, or scikit-learn can help with tokenization.

2. **Text Cleaning**:

   - Remove special characters, punctuation, and any noise from the text data. This helps in simplifying the text and making it more suitable for analysis.

3. **Stopword Removal**:

   - Remove common stopwords (e.g., "the," "and," "is") from the text data. NLTK and spaCy have built-in stopword lists.

4. **Lemmatization or Stemming**:

   - Reduce words to their base form. Lemmatization and stemming help in reducing word variations to their root forms. Libraries like NLTK and spaCy provide lemmatization and stemming capabilities.

5. **TF-IDF (Term Frequency-Inverse Document Frequency)**:

   - Calculate TF-IDF scores for words in the text data. This numerical representation helps in measuring the importance of words in a document relative to a collection of documents.

   ```python
   from sklearn.feature_extraction.text import TfidfVectorizer
   ```

tfidf_vectorizer = TfidfVectorizer()

tfidf_matrix = tfidf_vectorizer.fit_transform(text_data)

6. **Word Embeddings**:

- Utilize word embeddings such as Word2Vec, FastText, or GloVe to represent words as dense vectors, capturing semantic relationships between words.

```python
from gensim.models import Word2Vec
word2vec_model = Word2Vec(sentences, vector_size=100, window=5, min_count=1)
```

7. **Contextual Embeddings**:

- For more advanced chatbots, consider using contextual embeddings such as BERT embeddings or models like GPT-3 for understanding context and generating responses.

8. **Named Entity Recognition (NER)**:

- Identify and extract named entities (e.g., names, dates, locations) from text data using NER models. spaCy and NLTK provide NER capabilities.

```python
import spacy
nlp = spacy.load("en_core_web_sm")
```

```
doc = nlp(text_data)
entities = [(entity.text, entity.label_) for entity in doc.ents]
```

9. **Part-of-Speech Tagging**:

   - Tag words with their parts of speech (e.g., noun, verb, adjective). This information can be useful for understanding the structure of sentences.

```python
import nltk
nltk.download('averaged_perceptron_tagger')
tokens = nltk.word_tokenize(text_data)
pos_tags = nltk.pos_tag(tokens)
```

10. **Length of Text**:

    - Calculate the length of the input text. This can be a simple feature used to understand the user's query.

11. **Sentiment Analysis**:

    - Analyze the sentiment of the text using sentiment analysis tools to determine whether the user's input is positive, negative, or neutral.

12. **Dialogue Context**:

- Maintain and update a context vector or dialogue history to keep track of the conversation flow and context. This is crucial for maintaining context during multi-turn conversations.

CONCLUSION:

In conclusion, creating a chatbot using Python is a fascinating and versatile endeavor that can address a wide range of tasks and applications. Chatbots have become an integral part of modern business, customer service, and user interaction, offering the potential for automation, improved user experiences, and efficient communication. Here are some key takeaways:

1. **Versatility and Utility**: Python provides a versatile platform for chatbot development. You can build chatbots for various domains, including customer support, information retrieval, entertainment, and more.

2. **Diverse Tools and Libraries**: Python offers a rich ecosystem of NLP libraries, machine learning frameworks, and pre-trained models that simplify the implementation of chatbot features, including intent recognition, entity extraction, and response generation.

3. **Data-Driven Approach**: Successful chatbots often rely on data and machine learning techniques. Collecting and preprocessing data, training models, and fine-tuning algorithms are essential steps in building effective chatbots.

4. **Feature Engineering**: Feature engineering, which involves transforming and extracting relevant information from text data, plays a critical role in chatbot development. Techniques like tokenization, TF-IDF, word embeddings, and sentiment analysis are valuable in creating context-aware chatbots.

5. **Natural Language Understanding**: Chatbots must excel in natural language understanding to interpret user queries, recognize intents, and extract entities. NLP libraries like spaCy, NLTK, and pre-trained models like BERT and GPT-3 are instrumental in this process.

6. **User Experience**: Building an intuitive and user-friendly interface for chatbot interaction is key. Integration with messaging platforms or web applications enhances the user experience.

7. **Continuous Improvement**: Chatbots require ongoing monitoring and maintenance. User feedback, performance metrics, and regular updates are crucial for improving chatbot capabilities and adapting to changing user needs.

8. **Deployment Options**: Python chatbots can be deployed in various ways, including as web applications, APIs, or integrations with messaging platforms.

9. **Innovation and Potential**: With advancements in NLP, machine learning, and AI, the potential for chatbots to provide intelligent and human-like interactions is continually expanding. Innovations like generative models and natural language understanding models are shaping the future of chatbots.

In summary, chatbot development in Python represents an exciting and dynamic field with the potential to transform how we interact with technology and businesses. By leveraging the power of Python and the tools available, you can create chatbots that are not only functional but also provide valuable and engaging user experiences.