



i have a intents.json file. i wan tot create a simple chatbot using nltk. suggest me



思 Thought for 27 seconds ^

Okay, the user wants to create a simple chatbot using NLTK with their intents.json file. Let me think through how to approach this step by step.

First, I need to outline the main components needed for a basic chatbot. Typically, this involves loading and processing the intents, preprocessing the data, training a model to recognize patterns, and then creating a response mechanism.

Starting with the intents.json file. The user probably has a structure with tags, patterns, and responses. So, the first step is to read this JSON file. I'll need to use the json module in Python to load the data.

Next, preprocessing the data. For each pattern in the intents, the chatbot should process the text by tokenizing, removing punctuation, and maybe lemmatizing. NLTK's word\_tokenize and WordNetLemmatizer can help here. Also, converting everything to lowercase to ensure case insensitivity.

Then, preparing training data. The idea is to create a bag of words for each pattern and associate it with the corresponding intent tag. So, collect all words from all patterns, lemmatize them, and create a list of unique words. For each pattern, create a bag where each entry is 1 if the word is present, else 0. The output is a list where each entry corresponds to the intent tag (using one-hot encoding).

After that, building the neural network model. Since this is a simple chatbot, a feedforward neural network with a couple of layers should suffice. Using Keras with TensorFlow backend makes it straightforward. The input layer size matches the bag of words length, a hidden layer, then an output layer with softmax for probability distribution over intents.

Training the model on the prepared data. The model will learn to predict the correct intent based on the input bag of words. After training, save the model and the necessary preprocessing data (words, tags, etc.) for later use in predictions.

Once the model is trained, the next step is to implement the chat function. Here, the user inputs a sentence, which is processed into a bag of words. The model predicts the intent, and a response is randomly selected from the intents' responses.

Potential issues to consider: The user might not have all the required NLTK datasets (like punkt, wordnet), so the code should include downloading those. Also, the lemmatizer might need part-of-speech tags for better accuracy, but for simplicity, maybe it's omitted here. The model's training parameters (epochs, batch size) can be adjusted, but default values are a starting point.

Testing the chatbot with sample inputs to ensure it's working. The user should be informed to type 'quit' to exit the loop.

Finally, putting all these steps into code, making sure to handle the data loading, preprocessing, model training, and the chat interface. Also, mentioning possible improvements if the user wants to expand the project later.

I'll guide you through creating a simple chatbot using NLTK with your intents.json file. Here's a step-by-step implementation:

### 1. First, install required packages:

```
bash
```

Copy

```
pip install nltk tensorflow numpy
```

### 2. Here's the complete code: (Create a new Python file)

```
python
```

Copy

```
import json
import random
import numpy as np
import nltk
from nltk.stem import WordNetLemmatizer
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import SGD

# Download NLTK data
nltk.download('punkt')
nltk.download('wordnet')

# Initialize lemmatizer
lemmatizer = WordNetLemmatizer()

# Load intents file
with open('intents.json') as file:
    intents = json.load(file)

# Preprocess data
words = []
```

```

classes = []
documents = []
ignore_chars = ['?', '!', '.', ',']

for intent in intents['intents']:
    for pattern in intent['patterns']:
        # Tokenize words
        word_list = nltk.word_tokenize(pattern)
        words.extend(word_list)
        documents.append((word_list, intent['tag']))
        if intent['tag'] not in classes:
            classes.append(intent['tag'])

# Lemmatize and clean words
words = [lemmatizer.lemmatize(word.lower()) for word in words if word not in ignore_chars]
words = sorted(list(set(words)))
classes = sorted(list(set(classes)))

# Create training data
training = []
output_empty = [0] * len(classes)

for doc in documents:
    bag = []
    word_patterns = doc[0]
    word_patterns = [lemmatizer.lemmatize(word.lower()) for word in word_patterns]

    for word in words:
        bag.append(1) if word in word_patterns else bag.append(0)

    output_row = list(output_empty)
    output_row[classes.index(doc[1])] = 1
    training.append([bag, output_row])

# Shuffle and convert to array
random.shuffle(training)
training = np.array(training, dtype=object)

# Create train/test split
train_x = list(training[:, 0])
train_y = list(training[:, 1])

# Create model
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'))

```

```

# Compile model
sgd = SGD(learning_rate=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])

# Train model
model.fit(np.array(train_x), np.array(train_y), epochs=200, batch_size=5, verbose=1)

# Save model and preprocessing data (optional)
# model.save('chatbot_model.h5')
# import pickle
# pickle.dump({'words':words, 'classes':classes, 'train_x':train_x, 'train_y':train_y},
open("training_data.pkl", "wb"))

# Chat function
def chat():
    print("Bot is running! Type 'quit' to exit")
    while True:
        inp = input("You: ")
        if inp.lower() == 'quit':
            break

        # Process input
        inp_words = nltk.word_tokenize(inp)
        inp_words = [lemmatizer.lemmatize(word.lower()) for word in inp_words]

        # Create bag of words
        bag = [0] * len(words)
        for word in inp_words:
            if word in words:
                bag[words.index(word)] = 1

        # Predict class
        results = model.predict(np.array([bag]))[0]
        results_index = np.argmax(results)
        tag = classes[results_index]

        # Get response
        for intent in intents['intents']:
            if intent['tag'] == tag:
                responses = intent['responses']
                print(random.choice(responses))
                break

# Start chatting
chat()

```

3. Your **intents.json** should look like this (modify according to your needs):

json

Copy

```
{  
  "intents": [  
    {  
      "tag": "greeting",  
      "patterns": ["Hi", "Hello", "Hey", "How are you"],  
      "responses": ["Hello!", "Hi there!", "Good to see you!"]  
    },  
    {  
      "tag": "goodbye",  
      "patterns": ["Bye", "See you later", "Goodbye"],  
      "responses": ["Goodbye!", "See you later!", "Take care!"]  
    },  
    {  
      "tag": "thanks",  
      "patterns": ["Thanks", "Thank you", "Appreciate it"],  
      "responses": ["You're welcome!", "No problem!", "Happy to help!"]  
    }  
  ]  
}
```

#### 4. Key components explained:

- **Preprocessing:** Tokenizes and lemmatizes words
- **Training data:** Creates bag-of-words representations
- **Neural Network:** Simple 3-layer model for intent classification
- **Chat interface:** Handles user input and generates responses

#### 5. To improve:

- Add more training data in intents.json
- Implement better text preprocessing
- Add error handling
- Include entity recognition
- Use better model architecture (e.g., LSTM)
- Add context awareness

The chatbot will:

1. Load your intents from JSON
2. Preprocess and train a neural network
3. Use the trained model to predict intents

4. Return random responses from matching intents

To use it:

1. Save your intents.json
2. Run the Python script
3. Type messages to chat with the bot
4. Type 'quit' to exit

The model might take a few minutes to train depending on your dataset size. For production use, you'd want to save the trained model and preprocessing data rather than retraining every time.

instead of neural network can we use any machine learning alsorithm for intent classificaion

< 2 / 2 >



Yes, you can replace the neural network with a traditional machine learning algorithm for intent classification. A popular choice for text classification tasks is the **Naive Bayes classifier**, which works well for intent classification. Other options include **Support Vector Machines (SVM)**, **Logistic Regression**, or **Random Forest**.

Below is an updated version of the chatbot using **Naive Bayes** from the `scikit-learn` library instead of a neural network:

## Updated Code with Naive Bayes Classifier

```
python                                         Copy
import json
import random
import numpy as np
import nltk
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import make_pipeline

# Download NLTK data
nltk.download('punkt')
nltk.download('wordnet')
```

```

# Initialize lemmatizer
lemmatizer = WordNetLemmatizer()

# Load intents file
with open('intents.json') as file:
    intents = json.load(file)

# Preprocess data
words = []
classes = []
documents = []
ignore_chars = ['?', '!', '.', ',']

for intent in intents['intents']:
    for pattern in intent['patterns']:
        # Tokenize words
        word_list = nltk.word_tokenize(pattern)
        words.extend(word_list)
        documents.append(( ' '.join(word_list), intent['tag']))
        if intent['tag'] not in classes:
            classes.append(intent['tag'])

# Lemmatize and clean words
words = [lemmatizer.lemmatize(word.lower()) for word in words if word not in ignore_chars]
words = sorted(list(set(words)))
classes = sorted(list(set(classes)))

# Prepare training data
X = [] # Patterns
y = [] # Intents (tags)

for doc in documents:
    X.append(doc[0]) # Pattern
    y.append(doc[1]) # Tag

# Create a pipeline with CountVectorizer and Naive Bayes
model = make_pipeline(CountVectorizer(), MultinomialNB())

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

# Chat function
def chat():
    print("Bot is running! Type 'quit' to exit")
    while True:
        inp = input("You: ")
        if inp.lower() == 'quit':
            break

```

```

# Process input
inp_words = nltk.word_tokenize(inp)
inp_words = [lemmatizer.lemmatize(word.lower()) for word in inp_words]
processed_input = ' '.join(inp_words)

# Predict intent
predicted_tag = model.predict([processed_input])[0]

# Get response
for intent in intents['intents']:
    if intent['tag'] == predicted_tag:
        responses = intent['responses']
        print(random.choice(responses))
        break

# Start chatting
chat()

```

## Key Changes:

### 1. Replaced Neural Network with Naive Bayes:

- Used `scikit-learn`'s `MultinomialNB` for intent classification.
- Created a pipeline with `CountVectorizer` to convert text into numerical features.

### 2. Simplified Training:

- Instead of creating a bag-of-words manually, `CountVectorizer` handles it.
- The model is trained directly on the patterns and their corresponding tags.

### 3. Efficiency:

- Naive Bayes is lightweight and trains much faster than a neural network.
- Works well for small to medium-sized datasets.

---

## How It Works:

### 1. Preprocessing:

- Tokenizes and lemmatizes the input patterns.
- Converts patterns into a format suitable for machine learning.

### 2. Training:

- Uses `CountVectorizer` to convert text into numerical features (bag-of-words).
- Trains a Naive Bayes classifier on the processed data.

### 3. Prediction:

- Takes user input, preprocesses it, and predicts the intent using the trained model.
  - Selects a random response from the matching intent.
- 

Example `intents.json`:

```
json                                         Copy
{
  "intents": [
    {
      "tag": "greeting",
      "patterns": ["Hi", "Hello", "Hey", "How are you"],
      "responses": ["Hello!", "Hi there!", "Good to see you!"]
    },
    {
      "tag": "goodbye",
      "patterns": ["Bye", "See you later", "Goodbye"],
      "responses": ["Goodbye!", "See you later!", "Take care!"]
    },
    {
      "tag": "thanks",
      "patterns": ["Thanks", "Thank you", "Appreciate it"],
      "responses": ["You're welcome!", "No problem!", "Happy to help!"]
    }
  ]
}
```

---

Advantages of Using Naive Bayes:

- **Fast training and prediction.**
  - **Works well with small datasets.**
  - **Simple to implement and interpret.**
- 

To Improve:

1. **Add more training data** to improve accuracy.
2. **Use TF-IDF instead of CountVectorizer** for better feature representation.
3. **Try other algorithms** like SVM or Logistic Regression for comparison.
4. **Add entity recognition** for more advanced conversations.

This approach is lightweight and works well for simple chatbots. If your dataset grows or

you need more advanced features, you can switch back to neural networks or explore other ML algorithms.

use tfidf and svm for the above code

 Sure! Let's update the code to use **TF-IDF (Term Frequency-Inverse Document Frequency)** for feature extraction and **Support Vector Machine (SVM)** for intent classification. SVM is a powerful algorithm for text classification tasks and works well with TF-IDF features.

Here's the updated code:

## Updated Code with TF-IDF and SVM

```
python Copy
import json
import random
import nltk
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline

# Download NLTK data
nltk.download('punkt')
nltk.download('wordnet')

# Initialize lemmatizer
lemmatizer = WordNetLemmatizer()

# Load intents file
with open('intents.json') as file:
    intents = json.load(file)

# Preprocess data
words = []
classes = []
documents = []
ignore_chars = ['?', '!', '.', ',']

for intent in intents['intents']:
    for pattern in intent['patterns']:
```

```

# Tokenize words
word_list = nltk.word_tokenize(pattern)
words.extend(word_list)
documents.append(' '.join(word_list), intent['tag']))
if intent['tag'] not in classes:
    classes.append(intent['tag'])

# Lemmatize and clean words
words = [lemmatizer.lemmatize(word.lower()) for word in words if word not in ignore_chars]
words = sorted(list(set(words)))
classes = sorted(list(set(classes)))

# Prepare training data
X = [] # Patterns
y = [] # Intents (tags)

for doc in documents:
    X.append(doc[0]) # Pattern
    y.append(doc[1]) # Tag

# Create a pipeline with TF-IDF and SVM
model = make_pipeline(TfidfVectorizer(), SVC(kernel='linear', probability=True))

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

# Chat function
def chat():
    print("Bot is running! Type 'quit' to exit")
    while True:
        inp = input("You: ")
        if inp.lower() == 'quit':
            break

        # Process input
        inp_words = nltk.word_tokenize(inp)
        inp_words = [lemmatizer.lemmatize(word.lower()) for word in inp_words]
        processed_input = ' '.join(inp_words)

        # Predict intent
        predicted_tag = model.predict([processed_input])[0]

        # Get response
        for intent in intents['intents']:
            if intent['tag'] == predicted_tag:
                responses = intent['responses']
                print(random.choice(responses))
                break

```

```
# Start chatting
chat()
```

---

## Key Changes:

### 1. TF-IDF Vectorizer:

- Replaced `CountVectorizer` with `TfidfVectorizer` to better capture the importance of words in the patterns.
- TF-IDF considers both the frequency of a word in a document and its rarity across all documents.

### 2. SVM Classifier:

- Used `SVC` (Support Vector Classifier) with a **linear kernel** for intent classification.
- Set `probability=True` to enable probability estimates (useful for confidence scoring in future enhancements).

### 3. Pipeline:

- Combined `TfidfVectorizer` and `SVC` into a single pipeline for cleaner code and easier training.
- 

## How It Works:

### 1. Preprocessing:

- Tokenizes and lemmatizes the input patterns.
- Converts patterns into TF-IDF features.

### 2. Training:

- Trains an SVM model on the TF-IDF features and corresponding intent tags.

### 3. Prediction:

- Takes user input, preprocesses it, and predicts the intent using the trained SVM model.
  - Selects a random response from the matching intent.
- 

## Example `intents.json`:

```
json
```

```
Copy
```

```
{
  "intents": [
```

```
{  
    "tag": "greeting",  
    "patterns": ["Hi", "Hello", "Hey", "How are you"],  
    "responses": ["Hello!", "Hi there!", "Good to see you!"]  
},  
{  
    "tag": "goodbye",  
    "patterns": ["Bye", "See you later", "Goodbye"],  
    "responses": ["Goodbye!", "See you later!", "Take care!"]  
},  
{  
    "tag": "thanks",  
    "patterns": ["Thanks", "Thank you", "Appreciate it"],  
    "responses": ["You're welcome!", "No problem!", "Happy to help!"]  
}  
]  
}
```

---

## Advantages of Using TF-IDF and SVM:

### 1. TF-IDF:

- Better than bag-of-words for capturing word importance.
- Reduces the impact of common words (e.g., "the", "is") and emphasizes meaningful words.

### 2. SVM:

- Effective for high-dimensional data (like text).
- Works well with small to medium-sized datasets.
- Provides good generalization.

---

## To Improve:

### 1. Hyperparameter Tuning:

- Experiment with different kernels (e.g., `rbf`, `poly`) in SVM.
- Tune the `C` parameter for better performance.

### 2. Add Confidence Scoring:

- Use `model.predict_proba()` to get confidence scores for predictions and handle low-confidence cases.

### 3. Expand Dataset:

- Add more intents and patterns to improve the chatbot's understanding.

#### 4. Advanced Preprocessing:

- Remove stopwords, handle synonyms, or use stemming.

#### Example Output:

Copy

```
Bot is running! Type 'quit' to exit
You: Hi
Hello!
You: Thank you
You're welcome!
You: Bye
Goodbye!
```

This implementation is efficient and works well for small to medium-sized datasets. If your dataset grows, you can explore more advanced techniques like deep learning or ensemble methods.

dod we need to have a traingin gdataset for this ?



Yes, you **need a training dataset** for this approach. The training dataset is essential for teaching the chatbot how to map user inputs (patterns) to their corresponding intents (tags). In the code I provided, the training dataset is derived from the `intents.json` file, which contains examples of user inputs ( `patterns` ) and their associated intents ( `tags` ).

### Why Do You Need a Training Dataset?

#### 1. Supervised Learning:

- The chatbot uses a supervised machine learning approach (SVM with TF-IDF in this case).
- Supervised learning requires labeled data, where each input (pattern) is associated with a correct output (intent/tag).

#### 2. Intent Classification:

- The model learns to classify user inputs into predefined intents (e.g., "greeting", "goodbye", "thanks").

- Without labeled examples, the model cannot learn the relationship between inputs and outputs.

- Without labeled examples, the model cannot learn the relationship between inputs and intents.

### 3. Generalization:

- The training dataset helps the model generalize to new, unseen inputs that are similar to the training examples.

## What Does the Training Dataset Look Like?

The training dataset is typically structured in a JSON file (e.g., `intents.json`). Here's an example:

```
json                                         Copy
{
  "intents": [
    {
      "tag": "greeting",
      "patterns": ["Hi", "Hello", "Hey", "How are you"],
      "responses": ["Hello!", "Hi there!", "Good to see you!"]
    },
    {
      "tag": "goodbye",
      "patterns": ["Bye", "See you later", "Goodbye"],
      "responses": ["Goodbye!", "See you later!", "Take care!"]
    },
    {
      "tag": "thanks",
      "patterns": ["Thanks", "Thank you", "Appreciate it"],
      "responses": ["You're welcome!", "No problem!", "Happy to help!"]
    }
  ]
}
```

- `tag`: The intent label (e.g., "greeting").
- `patterns`: Example user inputs that belong to this intent.
- `responses`: Possible responses the chatbot can give for this intent.

## How Does the Training Dataset Work in the Code?

### 1. Preprocessing:

- The `patterns` are tokenized, lemmatized, and cleaned.
- The `tags` are used as labels for training.

### 2. Feature Extraction:

#### **2. Feature Extraction:**

- The `patterns` are converted into numerical features using TF-IDF.

#### **3. Training:**

- The SVM model learns to map the TF-IDF features to the corresponding `tags`.

#### **4. Prediction:**

- When a user inputs a message, it is preprocessed, converted into TF-IDF features, and classified into one of the intents.
- 

## What If You Don't Have a Training Dataset?

If you don't have a training dataset, you can:

#### **1. Create One Manually:**

- Define the intents (e.g., "greeting", "goodbye").
- Write example patterns for each intent.
- Add possible responses for each intent.

#### **2. Use a Predefined Dataset:**

- Use publicly available datasets for chatbot training, such as:
  - [ChatterBot Corpus](#)
  - [SNIPS NLU Dataset](#)
  - [ATIS Dataset](#)

#### **3. Use a Rule-Based Approach:**

- If you don't want to use machine learning, you can create a rule-based chatbot using keyword matching or regular expressions. However, this approach is less flexible and scalable.
- 

## Example of Creating a Training Dataset

Here's how you can create a simple `intents.json` file:

#### **1. Define the intents and their patterns:**

- **Intent:** `greeting`
  - Patterns: `["Hi", "Hello", "Hey", "How are you"]`
  - Responses: `["Hello!", "Hi there!", "Good to see you!"]`
- **Intent:** `goodbye`

- Patterns: ["Bye", "See you later", "Goodbye"]
- Responses: ["Goodbye!", "See you later!", "Take care!"]
- Intent: thanks
  - Patterns: ["Thanks", "Thank you", "Appreciate it"]
  - Responses: ["You're welcome!", "No problem!", "Happy to help!"]

2. Save it as `intents.json`:

```
json                                         Copy

{
  "intents": [
    {
      "tag": "greeting",
      "patterns": ["Hi", "Hello", "Hey", "How are you"],
      "responses": ["Hello!", "Hi there!", "Good to see you!"]
    },
    {
      "tag": "goodbye",
      "patterns": ["Bye", "See you later", "Goodbye"],
      "responses": ["Goodbye!", "See you later!", "Take care!"]
    },
    {
      "tag": "thanks",
      "patterns": ["Thanks", "Thank you", "Appreciate it"],
      "responses": ["You're welcome!", "No problem!", "Happy to help!"]
    }
  ]
}
```

3. Use this file in the code to train the chatbot.

---

## Can You Build a Chatbot Without a Training Dataset?

If you don't want to create or use a training dataset, you can build a **rule-based chatbot** using simple keyword matching. Here's an example:

```
python                                         Copy

import random

# Define rules
rules = {
  "greeting": ["hi", "hello", "hey"],
  "goodbye": ["bye", "goodbye", "see you"],
  "thanks": ["thanks", "thank you"]
```

```

}

# Define responses
responses = {
    "greeting": ["Hello!", "Hi there!", "Good to see you!"],
    "goodbye": ["Goodbye!", "See you later!", "Take care!"],
    "thanks": ["You're welcome!", "No problem!", "Happy to help!"]
}

# Chat function
def chat():
    print("Bot is running! Type 'quit' to exit")
    while True:
        inp = input("You: ").lower()
        if inp == 'quit':
            break

        # Match input to rules
        matched_intent = None
        for intent, keywords in rules.items():
            if any(keyword in inp for keyword in keywords):
                matched_intent = intent
                break

        # Get response
        if matched_intent:
            print(random.choice(responses[matched_intent]))
        else:
            print("I'm sorry, I don't understand.")

# Start chatting
chat()

```

This approach doesn't require a training dataset but is limited to predefined rules and keywords.

---

## Conclusion:

- **For machine learning-based chatbots:** You need a training dataset (e.g., `intents.json`).
- **For rule-based chatbots:** You don't need a training dataset, but the chatbot's capabilities will be limited.

If you're building a chatbot for the first time, start with a small training dataset and expand it as needed.

does intent looks like an excel file



No, **intents** are typically not stored in an Excel file. Instead, they are usually structured in a **JSON file** (like `intents.json`) because JSON is a flexible and widely used format for storing structured data, especially for chatbots and natural language processing (NLP) tasks.

However, if you have your intents in an Excel file, you can easily convert it into the required JSON format for use in your chatbot. Let me explain both scenarios:

## 1. Intents in JSON Format (Standard Approach)

The `intents.json` file typically looks like this:

```
json                                         Copy

{
  "intents": [
    {
      "tag": "greeting",
      "patterns": ["Hi", "Hello", "Hey", "How are you"],
      "responses": ["Hello!", "Hi there!", "Good to see you!"]
    },
    {
      "tag": "goodbye",
      "patterns": ["Bye", "See you later", "Goodbye"],
      "responses": ["Goodbye!", "See you later!", "Take care!"]
    },
    {
      "tag": "thanks",
      "patterns": ["Thanks", "Thank you", "Appreciate it"],
      "responses": ["You're welcome!", "No problem!", "Happy to help!"]
    }
  ]
}
```

- **tag** : Represents the intent (e.g., "greeting").
- **patterns** : Example user inputs that belong to this intent.
- **responses** : Possible responses the chatbot can give for this intent.

This format is easy to read, write, and process in Python using the `json` library.

## 2. Intents in Excel Format

If your intents are stored in an Excel file, they might look like this:

Tag	Patterns	Responses
greetin g	Hi, Hello, Hey, How are you	Hello!, Hi there!, Good to see you!
goodby e	Bye, See you later, Goodbye	Goodbye!, See you later!, Take care!
thanks	Thanks, Thank you, Appreciate it	You're welcome!, No problem!, Happy to help!

### How to Convert Excel to JSON

If your intents are in an Excel file, you can convert them into the required JSON format using Python. Here's how:

#### Step 1: Install Required Libraries

```
bash                                         Copy
pip install pandas openpyxl
```

#### Step 2: Convert Excel to JSON

Here's a Python script to convert an Excel file (`intents.xlsx`) into a JSON file (`intents.json`):

```
python                                         Copy
import pandas as pd

# Load the Excel file
excel_file = "intents.xlsx"
df = pd.read_excel(excel_file)

# Convert to JSON format
intents = {"intents": []}

for _, row in df.iterrows():
    intents["intents"].append(row.to_dict())
```

```

intent = {
    "tag": row["Tag"],
    "patterns": row["Patterns"].split(", "),
    "responses": row["Responses"].split(", ")
}
intents["intents"].append(intent)

# Save to JSON file
import json
with open("intents.json", "w") as file:
    json.dump(intents, file, indent=4)

print("Excel file converted to intents.json")

```

Input Excel File ( `intents.xlsx` ):

Tag	Patterns	Responses
greeting	Hi, Hello, Hey, How are you	Hello!, Hi there!, Good to see you!
goodbye	Bye, See you later, Goodbye	Goodbye!, See you later!, Take care!
thanks	Thanks, Thank you, Appreciate it	You're welcome!, No problem!, Happy to help!

Output JSON File ( `intents.json` ):

```

json
Copy

{
  "intents": [
    {
      "tag": "greeting",
      "patterns": ["Hi", "Hello", "Hey", "How are you"],
      "responses": ["Hello!", "Hi there!", "Good to see you!"]
    },
    {
      "tag": "goodbye",
      "patterns": ["Bye", "See you later", "Goodbye"],
      "responses": ["Goodbye!", "See you later!", "Take care!"]
    },
    {
      "tag": "thanks",
      "patterns": ["Thanks", "Thank you", "Appreciate it"],
      "responses": ["You're welcome!", "No problem!", "Happy to help!"]
    }
  ]
}
```

```
        }
    ]
}
```

---

## Why JSON is Preferred Over Excel for Intents

### 1. Ease of Use:

- JSON is lightweight and easy to read/write in Python.
- It is widely used in web development and APIs.

### 2. Structured Data:

- JSON allows nested structures, making it ideal for storing intents, patterns, and responses.

### 3. Compatibility:

- Most NLP libraries and frameworks (e.g., NLTK, TensorFlow, PyTorch) work seamlessly with JSON.

### 4. No External Dependencies:

- JSON files can be read directly using Python's built-in `json` library, whereas Excel files require additional libraries like `pandas` or `openpyxl`.

---

## Can You Use Excel Directly?

Yes, you can use the Excel file directly in your chatbot code without converting it to JSON. Here's how:

```
python
import pandas as pd

# Load the Excel file
excel_file = "intents.xlsx"
df = pd.read_excel(excel_file)

# Prepare training data
X = []
y = []

for _, row in df.iterrows():
    patterns = row["Patterns"].split(", ")
    responses = row["Responses"].split(", ")
    tag = row["Tag"]
```

Copy

```
for pattern in patterns:  
    X.append(pattern)  
    y.append(tag)  
  
# Now X and y can be used for training  
print("Training data:", X)  
print("Labels:", y)
```

However, this approach is less common because JSON is more suitable for structured data like intents.

---

## Conclusion:

- **Standard Format:** Use `intents.json` for storing intents, patterns, and responses.
- **Excel to JSON:** If your data is in Excel, convert it to JSON using Python.
- **Direct Excel Usage:** You can use Excel directly, but JSON is more efficient and widely used for chatbot development.

Let me know if you need further clarification!

i wan to chekc if my predicted intent vs real intent are same or not



To check if the **predicted intent** matches the **real intent**, you need to evaluate your chatbot's performance. This is typically done by comparing the predicted intents against a set of **ground truth labels** (real intents) using metrics like **accuracy**, **precision**, **recall**, or **F1-score**.

Here's how you can implement this:

---

## Steps to Compare Predicted vs Real Intents

### 1. Prepare a Test Dataset:

- Create a test dataset with input patterns and their corresponding real intents (ground truth labels).

### 2. Predict Intents:

- Use your trained model to predict intents for the test dataset.

### 3. Compare Predictions:

- Compare the predicted intents with the real intents.

#### 4. Calculate Metrics:

- Use evaluation metrics to measure the performance of your chatbot.

## Implementation

Here's the Python code to compare predicted intents with real intents:

```
python Copy

import json
import random
import nltk
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Download NLTK data
nltk.download('punkt')
nltk.download('wordnet')

# Initialize lemmatizer
lemmatizer = WordNetLemmatizer()

# Load intents file
with open('intents.json') as file:
    intents = json.load(file)

# Preprocess data
words = []
classes = []
documents = []
ignore_chars = ['?', '!', '.', ',']

for intent in intents['intents']:
    for pattern in intent['patterns']:
        # Tokenize words
        word_list = nltk.word_tokenize(pattern)
        words.extend(word_list)
        documents.append(( ' '.join(word_list), intent['tag']))
        if intent['tag'] not in classes:
            classes.append(intent['tag'])

# Lemmatize and clean words
words = [lemmatizer.lemmatize(word.lower()) for word in words if word not in ignore_cha
```

```
rs]

words = sorted(list(set(words)))
classes = sorted(list(set(classes)))

# Prepare training data
X_train = [] # Patterns
y_train = [] # Intents (tags)

for doc in documents:
    X_train.append(doc[0]) # Pattern
    y_train.append(doc[1]) # Tag

# Create a pipeline with TF-IDF and SVM
model = make_pipeline(TfidfVectorizer(), SVC(kernel='linear', probability=True))

# Train the model
model.fit(X_train, y_train)

# Prepare test data (replace this with your actual test data)
test_data = [
    ("Hi there!", "greeting"),
    ("Goodbye for now", "goodbye"),
    ("Thanks a lot", "thanks"),
    ("How are you?", "greeting"),
    ("See you later", "goodbye"),
    ("Thank you so much", "thanks"),
    ("Hello!", "greeting"),
    ("Bye", "goodbye"),
    ("Appreciate it", "thanks")
]

# Separate test inputs and real intents
X_test = [data[0] for data in test_data]
y_true = [data[1] for data in test_data]

# Predict intents
y_pred = model.predict(X_test)

# Compare predicted vs real intents
for i in range(len(X_test)):
    print(f"Input: {X_test[i]}")
    print(f"Predicted Intent: {y_pred[i]}")
    print(f"Real Intent: {y_true[i]}")
    print("---")

# Calculate accuracy
accuracy = accuracy_score(y_true, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Generate classification report
print("\nClassification Report:\n")
```

```
print("\nClassification Report: ")
print(classification_report(y_true, y_pred, target_names=classes))

# Generate confusion matrix
print("\nConfusion Matrix:")
print(confusion_matrix(y_true, y_pred, labels=classes))
```

---

## Explanation of the Code

### 1. Test Dataset:

- `test_data` contains input patterns and their corresponding real intents (ground truth labels).
- Replace this with your actual test dataset.

### 2. Predict Intents:

- The trained model (`model.predict`) is used to predict intents for the test inputs.

### 3. Compare Predictions:

- The predicted intents (`y_pred`) are compared with the real intents (`y_true`).

### 4. Evaluation Metrics:

- **Accuracy:** Percentage of correctly predicted intents.
- **Classification Report:** Includes precision, recall, and F1-score for each intent.
- **Confusion Matrix:** Shows how intents are being classified (useful for identifying misclassifications).

---

## Example Output

### Input vs Predicted vs Real Intents:

```
Copy

Input: Hi there!
Predicted Intent: greeting
Real Intent: greeting
---
Input: Goodbye for now
Predicted Intent: goodbye
Real Intent: goodbye
---
Input: Thanks a lot
Predicted Intent: thanks
Real Intent: thanks
---
```

Metrics:

Copy

Accuracy: 100.00%

Classification Report:

	precision	recall	f1-score	support
greeting	1.00	1.00	1.00	3
goodbye	1.00	1.00	1.00	3
thanks	1.00	1.00	1.00	3
accuracy			1.00	9
macro avg	1.00	1.00	1.00	9
weighted avg	1.00	1.00	1.00	9

Confusion Matrix:

```
[[3 0 0]
 [0 3 0]
 [0 0 3]]
```

## Key Metrics to Look For

### 1. Accuracy:

- Percentage of correctly predicted intents.
- High accuracy means your model is performing well.

### 2. Precision:

- Percentage of predicted intents that are correct.
- High precision means fewer false positives.

### 3. Recall:

- Percentage of real intents that are correctly predicted.
- High recall means fewer false negatives.

### 4. F1-Score:

- Harmonic mean of precision and recall.
- Useful for imbalanced datasets.

### 5. Confusion Matrix:

- Shows how intents are being classified.
- Diagonal values represent correct predictions

## Next Steps

### 1. Improve the Model:

- Add more training data to improve accuracy.
- Experiment with different algorithms (e.g., Logistic Regression, Random Forest).

### 2. Handle Misclassifications:

- Analyze the confusion matrix to identify which intents are being misclassified.
- Add more examples for those intents in the training data.

### 3. Test on Real-World Data:

- Evaluate the chatbot on real user inputs to ensure it performs well in production.

Let me know if you need further assistance!

i want chatbot to be more interactive and it should produce what i need. for example if i search for the particular toyota car, it should give me information of the toyota car or give me options to choose whdat toyota car



To make your chatbot more interactive and capable of providing specific information (e.g., details about a Toyota car), you need to enhance it with the following features:

### 1. Dynamic Responses:

- The chatbot should fetch or provide specific information based on user queries.

### 2. Context Awareness:

- The chatbot should remember the context of the conversation (e.g., if the user is asking about cars, it should stay in the "car" context).

### 3. Integration with External Data:

- Use APIs or databases to fetch real-time information (e.g., car details).

### 4. Options and Menus:

- Provide users with options to choose from (e.g., a list of Toyota car models).

### 5. Natural Language Understanding (NLU):

- Use NLU to extract entities (e.g., car brand, model) from user input.

## Updated Chatbot Code with Interactive Features

Here's how you can implement a more interactive chatbot:

```
python Copy
import json
import random
import nltk
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline

# Download NLTK data
nltk.download('punkt')
nltk.download('wordnet')

# Initialize lemmatizer
lemmatizer = WordNetLemmatizer()

# Load intents file
with open('intents.json') as file:
    intents = json.load(file)

# Preprocess data
words = []
classes = []
documents = []
ignore_chars = ['?', '!', '.', ',']

for intent in intents['intents']:
    for pattern in intent['patterns']:
        # Tokenize words
        word_list = nltk.word_tokenize(pattern)
        words.extend(word_list)
        documents.append(( ' '.join(word_list), intent['tag']))
        if intent['tag'] not in classes:
            classes.append(intent['tag'])

# Lemmatize and clean words
words = [lemmatizer.lemmatize(word.lower()) for word in words if word not in ignore_chars]
words = sorted(list(set(words)))
classes = sorted(list(set(classes)))

# Prepare training data
X_train = [] # Patterns
y_train = [] # Intents (tags)
```

```

for doc in documents:
    X_train.append(doc[0]) # Pattern
    y_train.append(doc[1]) # Tag

# Create a pipeline with TF-IDF and SVM
model = make_pipeline(TfidfVectorizer(), SVC(kernel='linear', probability=True))

# Train the model
model.fit(X_train, y_train)

# Car database (replace with actual API or database)
car_database = {
    "Toyota": {
        "Corolla": {
            "price": "$20,000",
            "engine": "1.8L 4-cylinder",
            "features": ["Bluetooth", "Backup Camera", "Apple CarPlay"]
        },
        "Camry": {
            "price": "$25,000",
            "engine": "2.5L 4-cylinder",
            "features": ["Leather Seats", "Sunroof", "Android Auto"]
        },
        "Rav4": {
            "price": "$28,000",
            "engine": "2.5L 4-cylinder",
            "features": ["AWD", "Touchscreen Display", "Blind Spot Monitor"]
        }
    }
}

# Chat function
def chat():
    print("Bot is running! Type 'quit' to exit")
    context = None # To keep track of the conversation context

    while True:
        inp = input("You: ")
        if inp.lower() == 'quit':
            break

        # Process input
        inp_words = nltk.word_tokenize(inp)
        inp_words = [lemmatizer.lemmatize(word.lower()) for word in inp_words]
        processed_input = ' '.join(inp_words)

        # Predict intent
        predicted_tag = model.predict([processed_input])[0]

```

```

# Handle context
if predicted_tag == "car_info":
    context = "car_info" # Set context to car_info
    print("Which Toyota car are you interested in? (Corolla, Camry, Rav4)")
    continue

if context == "car_info":
    car_model = None
    for word in inp_words:
        if word in car_database["Toyota"]:
            car_model = word
            break

    if car_model:
        car_details = car_database["Toyota"][car_model]
        print(f"Here are the details for Toyota {car_model}:")
        print(f"Price: {car_details['price']}")
        print(f"Engine: {car_details['engine']}")
        print(f"Features: {', '.join(car_details['features'])}")
        context = None # Reset context after providing information
    else:
        print("Sorry, I don't recognize that Toyota model. Please choose from Corolla, Camry, or Rav4.")
        continue

# Get response
for intent in intents['intents']:
    if intent['tag'] == predicted_tag:
        responses = intent['responses']
        print(random.choice(responses))
        break

# Start chatting
chat()

```

## Key Features Added

### 1. Dynamic Responses:

- The chatbot fetches car details from a `car_database` dictionary (replace this with an API or real database).

### 2. Context Awareness:

- The chatbot uses a `context` variable to remember the conversation flow (e.g., if the user is asking about cars).

### 3. Options and Menus:

- When the user asks for car information, the chatbot provides a list of Toyota

models to choose from.

#### 4. Entity Recognition:

- The chatbot identifies the car model from the user's input (e.g., "Tell me about the Toyota Camry").

---

### Example Conversation

```
Copy

Bot is running! Type 'quit' to exit
You: Hi
Hello!
You: I want to know about Toyota cars
Which Toyota car are you interested in? (Corolla, Camry, Rav4)
You: Camry
Here are the details for Toyota Camry:
Price: $25,000
Engine: 2.5L 4-cylinder
Features: Leather Seats, Sunroof, Android Auto
You: quit
```

---

### Enhancements You Can Add

#### 1. Integration with APIs:

- Replace the `car_database` dictionary with a real API (e.g., a car database API or web scraping).

#### 2. More Contexts:

- Add more contexts (e.g., booking a test drive, comparing cars).

#### 3. Natural Language Understanding (NLU):

- Use libraries like `spaCy` or `Rasa` for better entity recognition and intent classification.

#### 4. User Preferences:

- Remember user preferences (e.g., favorite car models) for personalized responses.

#### 5. Error Handling:

- Handle cases where the user provides invalid input (e.g., "I don't understand. Please choose from Corolla, Camry, or Rav4.").

```
json
Copy

{
  "intents": [
    {
      "tag": "greeting",
      "patterns": ["Hi", "Hello", "Hey", "How are you"],
      "responses": ["Hello!", "Hi there!", "Good to see you!"]
    },
    {
      "tag": "goodbye",
      "patterns": ["Bye", "See you later", "Goodbye"],
      "responses": ["Goodbye!", "See you later!", "Take care!"]
    },
    {
      "tag": "car_info",
      "patterns": ["Tell me about Toyota cars", "I want to know about Toyota", "Toyota car details"],
      "responses": ["Which Toyota car are you interested in? (Corolla, Camry, Rav4)"]
    }
  ]
}
```

This chatbot is now more interactive and can provide specific information based on user queries. Let me know if you need further assistance!

can we use database instead of api



Yes, you can absolutely use a **database** instead of an API to store and retrieve information (e.g., car details). Using a database is a great choice if you want to maintain full control over your data and avoid relying on external services.

In this case, you can use a **SQL database** (e.g., MySQL, PostgreSQL, SQLite) or a **NoSQL database** (e.g., MongoDB) to store the car details. Below, I'll show you how to integrate a **SQLite database** (a lightweight, file-based SQL database) into your chatbot.

## Steps to Use a Database

### 1. Set Up the Database:

- Create a database and a table to store car details.

- Populate the table with car information.

## 2. Connect to the Database:

- Use a Python library like `sqlite3` (built-in) or `SQLAlchemy` to interact with the database.

## 3. Fetch Data from the Database:

- Query the database to retrieve car details based on user input.

## 4. Integrate with the Chatbot:

- Replace the hardcoded `car_database` dictionary with data fetched from the database.

---

## Implementation with SQLite Database

### Step 1: Set Up the Database

1. Install SQLite (if not already installed). It comes pre-installed with Python.
2. Create a database and a table for car details.

Run the following Python script to create the database and table:

```
python Copy
import sqlite3

# Connect to SQLite database (or create it if it doesn't exist)
conn = sqlite3.connect('car_database.db')
cursor = conn.cursor()

# Create a table for car details
cursor.execute('''
CREATE TABLE IF NOT EXISTS cars (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    brand TEXT NOT NULL,
    model TEXT NOT NULL,
    price TEXT NOT NULL,
    engine TEXT NOT NULL,
    features TEXT NOT NULL
)
''')

# Insert sample data
cars_data = [
    ("Toyota", "Corolla", "$20,000", "1.8L 4-cylinder", "Bluetooth, Backup Camera, Apple CarPlay"),
    ("Ford", "Mustang", "$35,000", "5.0L V8", "Shelby GT500, Performance Pack, Navigation"),
    ("BMW", "M3", "$60,000", "3.0L Twin-Turbo", "M Performance, Carbon Fiber, Head-up Display"),
    ("Audi", "RS6", "$70,000", "4.0T V8", "Quattro, Dynamic Steering, Matrix LED"),
    ("Mercedes-Benz", "AMG G63", "$120,000", "6.3L V8", "AMG Performance, Carbon Ceramic Brakes, Night Package")
]
cursor.executemany("INSERT INTO cars VALUES (?, ?, ?, ?, ?)", cars_data)
conn.commit()
```

```
c_cars_data )  
        ("Toyota", "Camry", "$25,000", "2.5L 4-cylinder", "Leather Seats, Sunroof, Android  
Auto"),  
        ("Toyota", "Rav4", "$28,000", "2.5L 4-cylinder", "AWD, Touchscreen Display, Blind S  
pot Monitor")  
    ]  
  
    cursor.executemany(''  
    INSERT INTO cars (brand, model, price, engine, features)  
    VALUES (?, ?, ?, ?, ?)  
    ''', cars_data)  
  
    # Commit changes and close the connection  
    conn.commit()  
    conn.close()  
  
    print("Database and table created successfully!")
```

This script creates a SQLite database file named `car_database.db` and inserts sample car details into the `cars` table.

## Step 2: Update the Chatbot to Use the Database

Here's the updated chatbot code that fetches car details from the SQLite database:

```
python Copy  
  
import json  
import random  
import nltk  
import sqlite3  
from nltk.stem import WordNetLemmatizer  
from sklearn.feature_extraction.text import TfidfVectorizer  
from sklearn.svm import SVC  
from sklearn.pipeline import make_pipeline  
  
# Download NLTK data  
nltk.download('punkt')  
nltk.download('wordnet')  
  
# Initialize lemmatizer  
lemmatizer = WordNetLemmatizer()  
  
# Load intents file  
with open('intents.json') as file:  
    intents = json.load(file)  
  
# Preprocess data  
words = []
```

```

classes = []
documents = []
ignore_chars = ['?', '!', '.', ',']

for intent in intents['intents']:
    for pattern in intent['patterns']:
        # Tokenize words
        word_list = nltk.word_tokenize(pattern)
        words.extend(word_list)
        documents.append(' '.join(word_list), intent['tag']))
        if intent['tag'] not in classes:
            classes.append(intent['tag'])

# Lemmatize and clean words
words = [lemmatizer.lemmatize(word.lower()) for word in words if word not in ignore_chars]
words = sorted(list(set(words)))
classes = sorted(list(set(classes)))

# Prepare training data
X_train = [] # Patterns
y_train = [] # Intents (tags)

for doc in documents:
    X_train.append(doc[0]) # Pattern
    y_train.append(doc[1]) # Tag

# Create a pipeline with TF-IDF and SVM
model = make_pipeline(TfidfVectorizer(), SVC(kernel='linear', probability=True))

# Train the model
model.fit(X_train, y_train)

# Function to fetch car details from the database
def get_car_details(brand, model):
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    cursor.execute('''
        SELECT price, engine, features FROM cars
        WHERE brand = ? AND model = ?
    ''', (brand, model))

    result = cursor.fetchone()
    conn.close()

    if result:
        return {
            "price": result[0],
            "engine": result[1],

```

```

        "features": result[2].split(", ")
    }
else:
    return None

# Chat function
def chat():
    print("Bot is running! Type 'quit' to exit")
    context = None # To keep track of the conversation context

    while True:
        inp = input("You: ")
        if inp.lower() == 'quit':
            break

        # Process input
        inp_words = nltk.word_tokenize(inp)
        inp_words = [lemmatizer.lemmatize(word.lower()) for word in inp_words]
        processed_input = ' '.join(inp_words)

        # Predict intent
        predicted_tag = model.predict([processed_input])[0]

        # Handle context
        if predicted_tag == "car_info":
            context = "car_info" # Set context to car_info
            print("Which Toyota car are you interested in? (Corolla, Camry, Rav4)")
            continue

        if context == "car_info":
            car_model = None
            for word in inp_words:
                if word in ["corolla", "camry", "rav4"]:
                    car_model = word.capitalize()
                    break

            if car_model:
                car_details = get_car_details("Toyota", car_model)
                if car_details:
                    print(f"Here are the details for Toyota {car_model}:")
                    print(f"Price: {car_details['price']}"))
                    print(f"Engine: {car_details['engine']}"))
                    print(f"Features: {', '.join(car_details['features'])}")
                else:
                    print("Sorry, I couldn't find details for that car.")
                context = None # Reset context after providing information
            else:
                print("Sorry, I don't recognize that Toyota model. Please choose from Corolla, Camry, or Rav4.")
                continue

```

```
# Get response
for intent in intents['intents']:
    if intent['tag'] == predicted_tag:
        responses = intent['responses']
        print(random.choice(responses))
        break

# Start chatting
chat()
```

## Key Changes

### 1. Database Integration:

- The `get_car_details` function connects to the SQLite database and fetches car details based on the brand and model.

### 2. Dynamic Data Fetching:

- Instead of using a hardcoded dictionary, the chatbot now retrieves car details from the database.

### 3. Error Handling:

- If the car model is not found in the database, the chatbot informs the user.

## Example Conversation

Copy

```
Bot is running! Type 'quit' to exit
You: Hi
Hello!
You: Tell me about Toyota cars
Which Toyota car are you interested in? (Corolla, Camry, Rav4)
You: Camry
Here are the details for Toyota Camry:
Price: $25,000
Engine: 2.5L 4-cylinder
Features: Leather Seats, Sunroof, Android Auto
You: quit
```

## Advantages of Using a Database

### 1. Scalability:

- You can store and manage large amounts of data efficiently.

## 2. Flexibility:

- Easily update or add new car details without modifying the chatbot code.

## 3. Persistence:

- Data is stored permanently and can be accessed across multiple sessions.

## 4. Security:

- You can control access to the database and implement security measures.
- 

## Next Steps

### 1. Expand the Database:

- Add more car brands and models to the database.

### 2. Add More Features:

- Allow users to compare cars, book test drives, or find dealerships.

### 3. Use a More Advanced Database:

- Switch to a more robust database like MySQL or PostgreSQL if needed.

### 4. Improve Error Handling:

- Handle cases where the database connection fails or queries return no results.

Let me know if you need further assistance!

cann you expland database according to ou and generate an intent json out of it



Certainly! Let's expand the database to include more car brands and models, and then generate an `intents.json` file based on the data in the database. This will make the chatbot more dynamic and scalable.

---

## Step 1: Expand the Database

We'll add more car brands and models to the SQLite database. Here's the updated script to create and populate the database:

```
python
```

Copy

```
import sqlite3
```

```
# Connect to SQLite database (or create it if it doesn't exist)
conn = sqlite3.connect('car_database.db')
cursor = conn.cursor()

# Create a table for car details
cursor.execute('''
CREATE TABLE IF NOT EXISTS cars (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    brand TEXT NOT NULL,
    model TEXT NOT NULL,
    price TEXT NOT NULL,
    engine TEXT NOT NULL,
    features TEXT NOT NULL
)
'''')

# Insert sample data
cars_data = [
    ("Toyota", "Corolla", "$20,000", "1.8L 4-cylinder", "Bluetooth, Backup Camera, Apple CarPlay"),
    ("Toyota", "Camry", "$25,000", "2.5L 4-cylinder", "Leather Seats, Sunroof, Android Auto"),
    ("Toyota", "Rav4", "$28,000", "2.5L 4-cylinder", "AWD, Touchscreen Display, Blind Spot Monitor"),
    ("Honda", "Civic", "$22,000", "2.0L 4-cylinder", "Bluetooth, LaneWatch, Apple CarPlay"),
    ("Honda", "Accord", "$27,000", "1.5L Turbo 4-cylinder", "Leather Seats, Sunroof, Android Auto"),
    ("Ford", "Mustang", "$35,000", "5.0L V8", "Leather Seats, Premium Audio, Performance Package"),
    ("Ford", "F-150", "$40,000", "3.5L EcoBoost V6", "Towing Package, Touchscreen Display, Android Auto"),
    ("Tesla", "Model 3", "$45,000", "Electric", "Autopilot, Touchscreen Display, Over-the-Air Updates"),
    ("Tesla", "Model S", "$80,000", "Electric", "Ludicrous Mode, Autopilot, Premium Interior")
]

cursor.executemany('''
INSERT INTO cars (brand, model, price, engine, features)
VALUES (?, ?, ?, ?, ?)
''', cars_data)

# Commit changes and close the connection
conn.commit()
conn.close()

print("Database and table created successfully!")
```

This script adds cars from **Toyota**, **Honda**, **Ford**, and **Tesla** to the database.

---

## Step 2: Generate `intents.json` from the Database

Now, let's generate an `intents.json` file dynamically based on the data in the database. This will allow the chatbot to handle queries for all car brands and models in the database.

Here's the Python script to generate the `intents.json` file:

```
python Copy
import sqlite3
import json

# Connect to the SQLite database
conn = sqlite3.connect('car_database.db')
cursor = conn.cursor()

# Fetch all car brands and models from the database
cursor.execute('SELECT DISTINCT brand FROM cars')
brands = [row[0] for row in cursor.fetchall()]

# Generate intents dynamically
intents = {
    "intents": [
        {
            "tag": "greeting",
            "patterns": ["Hi", "Hello", "Hey", "How are you"],
            "responses": ["Hello!", "Hi there!", "Good to see you!"]
        },
        {
            "tag": "goodbye",
            "patterns": ["Bye", "See you later", "Goodbye"],
            "responses": ["Goodbye!", "See you later!", "Take care!"]
        },
        {
            "tag": "car_info",
            "patterns": ["Tell me about cars", "I want to know about cars", "Car details"],
            "responses": ["Which car brand are you interested in? (Toyota, Honda, Ford, Tesla)"]
        }
    ]
}

# Add intents for each car brand
for brand in brands:
    intent = {
        "tag": f"{brand.lower()}_info",
```

```

        "patterns": [
            f"Tell me about {brand} cars",
            f"I want to know about {brand}",
            f"{brand} car details"
        ],
        "responses": [f"Which {brand} model are you interested in?"]
    }
intents["intents"].append(intent)

# Save intents to a JSON file
with open('intents.json', 'w') as file:
    json.dump(intents, file, indent=4)

print("intents.json file generated successfully!")

```

## Generated `intents.json` File

The script will generate an `intents.json` file like this:

```

json                                         Copy

{
  "intents": [
    {
      "tag": "greeting",
      "patterns": ["Hi", "Hello", "Hey", "How are you"],
      "responses": ["Hello!", "Hi there!", "Good to see you!"]
    },
    {
      "tag": "goodbye",
      "patterns": ["Bye", "See you later", "Goodbye"],
      "responses": ["Goodbye!", "See you later!", "Take care!"]
    },
    {
      "tag": "car_info",
      "patterns": ["Tell me about cars", "I want to know about cars", "Car details"],
      "responses": ["Which car brand are you interested in? (Toyota, Honda, Ford, Tesla)"]
    },
    {
      "tag": "toyota_info",
      "patterns": ["Tell me about Toyota cars", "I want to know about Toyota", "Toyota car details"],
      "responses": ["Which Toyota model are you interested in?"]
    },
    {
      "tag": "honda_info",
      "patterns": ["Tell me about Honda cars", "I want to know about Honda", "Hon

```

```
        "patterns": ["Tell me about Honda cars", "I want to know about Honda", "How  
da car details"],  
        "responses": ["Which Honda model are you interested in?"]  
    },  
    {  
        "tag": "ford_info",  
        "patterns": ["Tell me about Ford cars", "I want to know about Ford", "Ford  
car details"],  
        "responses": ["Which Ford model are you interested in?"]  
    },  
    {  
        "tag": "tesla_info",  
        "patterns": ["Tell me about Tesla cars", "I want to know about Tesla", "Tes  
la car details"],  
        "responses": ["Which Tesla model are you interested in?"]  
    }  
]
```

## Step 3: Update the Chatbot to Handle Dynamic Intents

Now that the `intents.json` file is generated dynamically, you can update the chatbot to handle queries for all car brands and models in the database.

Here's the updated chatbot code:

```
python Copy  
  
import json  
import random  
import nltk  
import sqlite3  
from nltk.stem import WordNetLemmatizer  
from sklearn.feature_extraction.text import TfidfVectorizer  
from sklearn.svm import SVC  
from sklearn.pipeline import make_pipeline  
  
# Download NLTK data  
nltk.download('punkt')  
nltk.download('wordnet')  
  
# Initialize lemmatizer  
lemmatizer = WordNetLemmatizer()  
  
# Load intents file  
with open('intents.json') as file:  
    intents = json.load(file)  
  
# Preprocess data
```

```

words = []
classes = []
documents = []
ignore_chars = ['?', '!', '.', ',']

for intent in intents['intents']:
    for pattern in intent['patterns']:
        # Tokenize words
        word_list = nltk.word_tokenize(pattern)
        words.extend(word_list)
        documents.append(( ' '.join(word_list), intent['tag']))
        if intent['tag'] not in classes:
            classes.append(intent['tag'])

# Lemmatize and clean words
words = [lemmatizer.lemmatize(word.lower()) for word in words if word not in ignore_chars]
words = sorted(list(set(words)))
classes = sorted(list(set(classes)))

# Prepare training data
X_train = [] # Patterns
y_train = [] # Intents (tags)

for doc in documents:
    X_train.append(doc[0]) # Pattern
    y_train.append(doc[1]) # Tag

# Create a pipeline with TF-IDF and SVM
model = make_pipeline(TfidfVectorizer(), SVC(kernel='linear', probability=True))

# Train the model
model.fit(X_train, y_train)

# Function to fetch car details from the database
def get_car_details(brand, model):
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    cursor.execute('''
    SELECT price, engine, features FROM cars
    WHERE brand = ? AND model = ?
    ''', (brand, model))

    result = cursor.fetchone()
    conn.close()

    if result:
        return {
            "price": result[0],

```

```
        "engine": result[1],
        "features": result[2].split(", ")
    }
else:
    return None

# Chat function
def chat():
    print("Bot is running! Type 'quit' to exit")
    context = None # To keep track of the conversation context
    current_brand = None # To store the current car brand

    while True:
        inp = input("You: ")
        if inp.lower() == 'quit':
            break

        # Process input
        inp_words = nltk.word_tokenize(inp)
        inp_words = [lemmatizer.lemmatize(word.lower()) for word in inp_words]
        processed_input = ' '.join(inp_words)

        # Predict intent
        predicted_tag = model.predict([processed_input])[0]

        # Handle context
        if predicted_tag.endswith("_info"):
            context = "car_info"
            current_brand = predicted_tag.replace("_info", "").capitalize()
            print(f"Which {current_brand} model are you interested in?")
            continue

        if context == "car_info":
            car_model = None
            cursor = conn.cursor()
            cursor.execute('''
                SELECT model FROM cars
                WHERE brand = ?
            ''', (current_brand,))
            models = [row[0].lower() for row in cursor.fetchall()]
            conn.close()

            for word in inp_words:
                if word in models:
                    car_model = word.capitalize()
                    break

            if car_model:
                car_details = get_car_details(current_brand, car_model)
                if car_details:
```

```

        print(f"Here are the details for {current_brand} {car_model}:")
        print(f"Price: {car_details['price']}") 
        print(f"Engine: {car_details['engine']}") 
        print(f"Features: {', '.join(car_details['features'])}") 
    else:
        print("Sorry, I couldn't find details for that car.")
    context = None # Reset context after providing information
else:
    print(f"Sorry, I don't recognize that {current_brand} model. Please choose from the available models.")
    continue

# Get response
for intent in intents['intents']:
    if intent['tag'] == predicted_tag:
        responses = intent['responses']
        print(random.choice(responses))
        break

# Start chatting
chat()

```

## Key Features of the Updated Chatbot

### 1. Dynamic Intents:

- The chatbot can handle queries for any car brand and model in the database.

### 2. Context Awareness:

- The chatbot remembers the current car brand and asks for the model.

### 3. Database Integration:

- Car details are fetched dynamically from the database.

### 4. Scalability:

- You can add more car brands and models to the database without modifying the chatbot code.

## Example Conversation

Copy

```

Bot is running! Type 'quit' to exit
You: Hi
Hello!
You: Tell me about Tesla cars
Which Tesla model are you interested in?

```

```
You: Model 3
Here are the details for Tesla Model 3:
Price: $45,000
Engine: Electric
Features: Autopilot, Touchscreen Display, Over-the-Air Updates
You: quit
```

This approach makes your chatbot highly scalable and dynamic. Let me know if you need further assistance!

thanks for this

 You're very welcome! 😊 I'm glad I could help you build a more interactive and dynamic chatbot. If you have any more questions, need further enhancements, or want to explore additional features (like integrating APIs, improving the UI, or adding more advanced NLP capabilities), feel free to ask.

Good luck with your chatbot project, and happy coding! 🚀🤖

Let me know if you need anything else! 😊

 if it is asking me the what models then first it shoul d give a=me the avaible models also so that i can select tit and bot can asnwer it

 Absolutely! That's a great idea. The chatbot should **list the available models** for the selected brand so the user can easily choose one. This makes the interaction more user-friendly and ensures the user knows what options are available.

Let's update the chatbot to **display the available models** when the user selects a brand. Here's how we can do it:

## Updated Chatbot Code with Model Listing

```
python
```

Copy

```
import json
import random
```

```
import random
import nltk
import sqlite3
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline

# Download NLTK data
nltk.download('punkt')
nltk.download('wordnet')

# Initialize lemmatizer
lemmatizer = WordNetLemmatizer()

# Load intents file
with open('intents.json') as file:
    intents = json.load(file)

# Preprocess data
words = []
classes = []
documents = []
ignore_chars = ['?', '!', '.', ',']

for intent in intents['intents']:
    for pattern in intent['patterns']:
        # Tokenize words
        word_list = nltk.word_tokenize(pattern)
        words.extend(word_list)
        documents.append(' '.join(word_list), intent['tag']))
        if intent['tag'] not in classes:
            classes.append(intent['tag'])

# Lemmatize and clean words
words = [lemmatizer.lemmatize(word.lower()) for word in words if word not in ignore_chars]
words = sorted(list(set(words)))
classes = sorted(list(set(classes)))

# Prepare training data
X_train = [] # Patterns
y_train = [] # Intents (tags)

for doc in documents:
    X_train.append(doc[0]) # Pattern
    y_train.append(doc[1]) # Tag

# Create a pipeline with TF-IDF and SVM
model = make_pipeline(TfidfVectorizer(), SVC(kernel='linear', probability=True))
```

```
# Train the model
model.fit(X_train, y_train)

# Function to fetch car details from the database
def get_car_details(brand, model):
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    cursor.execute('''
        SELECT price, engine, features FROM cars
        WHERE brand = ? AND model = ?
    ''', (brand, model))

    result = cursor.fetchone()
    conn.close()

    if result:
        return {
            "price": result[0],
            "engine": result[1],
            "features": result[2].split(", ")
        }
    else:
        return None

# Function to fetch available models for a brand
def get_available_models(brand):
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    cursor.execute('''
        SELECT model FROM cars
        WHERE brand = ?
    ''', (brand,))

    models = [row[0] for row in cursor.fetchall()]
    conn.close()

    return models

# Chat function
def chat():
    print("Bot is running! Type 'quit' to exit")
    context = None # To keep track of the conversation context
    current_brand = None # To store the current car brand

    while True:
        inp = input("You: ")
        if inp.lower() == 'quit':
            break
```

```

# Process input
inp_words = nltk.word_tokenize(inp)
inp_words = [lemmatizer.lemmatize(word.lower()) for word in inp_words]
processed_input = ' '.join(inp_words)

# Predict intent
predicted_tag = model.predict([processed_input])[0]

# Handle context
if predicted_tag.endswith("_info"):
    context = "car_info"
    current_brand = predicted_tag.replace("_info", "").capitalize()
    available_models = get_available_models(current_brand)
    print(f"Which {current_brand} model are you interested in? Available models: {', '.join(available_models)}")
    continue

if context == "car_info":
    car_model = None
    available_models = get_available_models(current_brand)
    for word in inp_words:
        if word.capitalize() in available_models:
            car_model = word.capitalize()
            break

    if car_model:
        car_details = get_car_details(current_brand, car_model)
        if car_details:
            print(f"Here are the details for {current_brand} {car_model}:")
            print(f"Price: {car_details['price']} ")
            print(f"Engine: {car_details['engine']} ")
            print(f"Features: {', '.join(car_details['features'])}")
        else:
            print("Sorry, I couldn't find details for that car.")
        context = None # Reset context after providing information
    else:
        print(f"Sorry, I don't recognize that {current_brand} model. Please choose from: {', '.join(available_models)}")
        continue

# Get response
for intent in intents['intents']:
    if intent['tag'] == predicted_tag:
        responses = intent['responses']
        print(random.choice(responses))
        break

# Start chatting
chat()

```

## Key Changes

### 1. `get_available_models` Function:

- This function fetches the available models for a given brand from the database.

### 2. Display Available Models:

- When the user selects a brand, the chatbot lists the available models for that brand.
- Example: Which Toyota model are you interested in? Available models:  
Corolla, Camry, Rav4

### 3. Improved Error Handling:

- If the user enters an invalid model, the chatbot reminds them of the available models.

## Example Conversation

Copy

```
Bot is running! Type 'quit' to exit
You: Hi
Hello!
You: Tell me about Toyota cars
Which Toyota model are you interested in? Available models: Corolla, Camry, Rav4
You: Camry
Here are the details for Toyota Camry:
Price: $25,000
Engine: 2.5L 4-cylinder
Features: Leather Seats, Sunroof, Android Auto
You: Tell me about Tesla cars
Which Tesla model are you interested in? Available models: Model 3, Model S
You: Model X
Sorry, I don't recognize that Tesla model. Please choose from: Model 3, Model S
You: quit
```

## Benefits of This Update

### 1. User-Friendly:

- Users know exactly what options are available, making the interaction smoother.

### 2. Error Prevention:

- By listing available models, the chatbot reduces the chance of users entering

- By listing available models, the chatbot reduces the chance of users entering invalid inputs.

### 3. Dynamic and Scalable:

- The chatbot dynamically fetches and displays models from the database, so it works even if you add new models later.
- 

## Next Steps

### 1. Add More Brands and Models:

- Expand the database with more car brands and models.

### 2. Improve UI:

- If you're building a web-based chatbot, you can display the available models as clickable buttons for better usability.

### 3. Add More Features:

- Allow users to compare cars, book test drives, or find dealerships.

### 4. Enhance Error Handling:

- Handle cases where the database connection fails or queries return no results.
- 

Let me know if you need further assistance or enhancements! 😊

should give me option s for the car band also



Absolutely! If you want the chatbot to **first list the available car brands** and then **list the models for the selected brand**, we can enhance the interaction further. This way, the user can first choose a brand and then select a model from the available options.

Here's how we can implement this:

---

## Updated Chatbot Code with Brand and Model Listing

python

Copy

```
import json
import random
import nltk
import sqlite3
```

```
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline

# Download NLTK data
nltk.download('punkt')
nltk.download('wordnet')

# Initialize lemmatizer
lemmatizer = WordNetLemmatizer()

# Load intents file
with open('intents.json') as file:
    intents = json.load(file)

# Preprocess data
words = []
classes = []
documents = []
ignore_chars = ['?', '!', '.', ',']

for intent in intents['intents']:
    for pattern in intent['patterns']:
        # Tokenize words
        word_list = nltk.word_tokenize(pattern)
        words.extend(word_list)
        documents.append(( ' '.join(word_list), intent['tag']))
        if intent['tag'] not in classes:
            classes.append(intent['tag'])

# Lemmatize and clean words
words = [lemmatizer.lemmatize(word.lower()) for word in words if word not in ignore_chars]
words = sorted(list(set(words)))
classes = sorted(list(set(classes)))

# Prepare training data
X_train = [] # Patterns
y_train = [] # Intents (tags)

for doc in documents:
    X_train.append(doc[0]) # Pattern
    y_train.append(doc[1]) # Tag

# Create a pipeline with TF-IDF and SVM
model = make_pipeline(TfidfVectorizer(), SVC(kernel='linear', probability=True))

# Train the model
model.fit(X_train, y_train)
```

```
# Function to fetch car details from the database
def get_car_details(brand, model):
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    cursor.execute('''
        SELECT price, engine, features FROM cars
        WHERE brand = ? AND model = ?
    ''', (brand, model))

    result = cursor.fetchone()
    conn.close()

    if result:
        return {
            "price": result[0],
            "engine": result[1],
            "features": result[2].split(", ")
        }
    else:
        return None

# Function to fetch available brands from the database
def get_available_brands():
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    cursor.execute('''
        SELECT DISTINCT brand FROM cars
    ''')

    brands = [row[0] for row in cursor.fetchall()]
    conn.close()

    return brands

# Function to fetch available models for a brand
def get_available_models(brand):
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    cursor.execute('''
        SELECT model FROM cars
        WHERE brand = ?
    ''', (brand,))

    models = [row[0] for row in cursor.fetchall()]
    conn.close()
```

```
return models

# Chat function
def chat():
    print("Bot is running! Type 'quit' to exit")
    context = None # To keep track of the conversation context
    current_brand = None # To store the current car brand

    while True:
        inp = input("You: ")
        if inp.lower() == 'quit':
            break

        # Process input
        inp_words = nltk.word_tokenize(inp)
        inp_words = [lemmatizer.lemmatize(word.lower()) for word in inp_words]
        processed_input = ' '.join(inp_words)

        # Predict intent
        predicted_tag = model.predict([processed_input])[0]

        # Handle context
        if predicted_tag == "car_info":
            context = "brand_selection"
            available_brands = get_available_brands()
            print(f"Which car brand are you interested in? Available brands: {', '.join(available_brands)}")
            continue

        if context == "brand_selection":
            selected_brand = None
            available_brands = get_available_brands()
            for word in inp_words:
                if word.capitalize() in available_brands:
                    selected_brand = word.capitalize()
                    break

            if selected_brand:
                context = "model_selection"
                current_brand = selected_brand
                available_models = get_available_models(current_brand)
                print(f"Which {current_brand} model are you interested in? Available models: {', '.join(available_models)}")
            else:
                print(f"Sorry, I don't recognize that brand. Please choose from: {', '.join(available_brands)}")
                continue

        if context == "model_selection":
            car_model = None
```

```

available_models = get_available_models(current_brand)
for word in inp_words:
    if word.capitalize() in available_models:
        car_model = word.capitalize()
        break

if car_model:
    car_details = get_car_details(current_brand, car_model)
    if car_details:
        print(f"Here are the details for {current_brand} {car_model}:")
        print(f"Price: {car_details['price']} ")
        print(f"Engine: {car_details['engine']} ")
        print(f"Features: {', '.join(car_details['features'])}")
    else:
        print("Sorry, I couldn't find details for that car.")
    context = None # Reset context after providing information
else:
    print(f"Sorry, I don't recognize that {current_brand} model. Please choose from: {', '.join(available_models)}")
    continue

# Get response
for intent in intents['intents']:
    if intent['tag'] == predicted_tag:
        responses = intent['responses']
        print(random.choice(responses))
        break

# Start chatting
chat()

```

## Key Changes

### 1. `get_available_brands` Function:

- Fetches the list of available car brands from the database.

### 2. Brand Selection Context:

- When the user asks for car information, the chatbot first lists the available brands.
- Example: Which car brand are you interested in? Available brands: Toyota, Honda, Ford, Tesla

### 3. Model Selection Context:

- After the user selects a brand, the chatbot lists the available models for that brand.
- Example: Which Toyota model are you interested in? Available models: Corolla, Camry, Rav4

#### 4. Improved Error Handling:

- If the user enters an invalid brand or model, the chatbot reminds them of the available options.
- 

#### Example Conversation

Copy

```
Bot is running! Type 'quit' to exit
You: Hi
Hello!
You: Tell me about cars
Which car brand are you interested in? Available brands: Toyota, Honda, Ford, Tesla
You: Toyota
Which Toyota model are you interested in? Available models: Corolla, Camry, Rav4
You: Camry
Here are the details for Toyota Camry:
Price: $25,000
Engine: 2.5L 4-cylinder
Features: Leather Seats, Sunroof, Android Auto
You: Tell me about Tesla
Which Tesla model are you interested in? Available models: Model 3, Model S
You: Model X
Sorry, I don't recognize that Tesla model. Please choose from: Model 3, Model S
You: quit
```

---

#### Benefits of This Update

##### 1. User-Friendly Interaction:

- The chatbot guides the user step-by-step, first listing brands and then models.

##### 2. Error Prevention:

- By listing available options, the chatbot reduces the chance of invalid inputs.

##### 3. Dynamic and Scalable:

- The chatbot dynamically fetches brands and models from the database, so it works even if you add new data later.
- 

#### Next Steps

##### 1. Add More Brands and Models:

- Expand the database with more car brands and models.

##### 2. Improve UI:

- If you're building a web-based chatbot, you can display the available brands and models as clickable buttons for better usability.

### 3. Add More Features:

- Allow users to compare cars, book test drives, or find dealerships.

### 4. Enhance Error Handling:

- Handle cases where the database connection fails or queries return no results.

---

Let me know if you need further assistance or enhancements! 😊

can we check if it can reset after giving details and ask again if i want to check other brand cars

Yes, we can definitely add a feature to **reset the conversation** after providing car details and **ask the user if they want to check another brand or model**. This makes the chatbot more interactive and user-friendly.

Here's how we can implement this:

---

### Updated Chatbot Code with Reset Option

```
python                                         Copy

import json
import random
import nltk
import sqlite3
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline

# Download NLTK data
nltk.download('punkt')
nltk.download('wordnet')

# Initialize lemmatizer
lemmatizer = WordNetLemmatizer()

# Load intents file
# ... (intents file content)
```

```

with open('intents.json') as file:
    intents = json.load(file)

# Preprocess data
words = []
classes = []
documents = []
ignore_chars = ['?', '!', '.', ',']

for intent in intents['intents']:
    for pattern in intent['patterns']:
        # Tokenize words
        word_list = nltk.word_tokenize(pattern)
        words.extend(word_list)
        documents.append(( ' '.join(word_list), intent['tag']))
        if intent['tag'] not in classes:
            classes.append(intent['tag'])

# Lemmatize and clean words
words = [lemmatizer.lemmatize(word.lower()) for word in words if word not in ignore_chars]
words = sorted(list(set(words)))
classes = sorted(list(set(classes)))

# Prepare training data
X_train = [] # Patterns
y_train = [] # Intents (tags)

for doc in documents:
    X_train.append(doc[0]) # Pattern
    y_train.append(doc[1]) # Tag

# Create a pipeline with TF-IDF and SVM
model = make_pipeline(TfidfVectorizer(), SVC(kernel='linear', probability=True))

# Train the model
model.fit(X_train, y_train)

# Function to fetch car details from the database
def get_car_details(brand, model):
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    cursor.execute('''
        SELECT price, engine, features FROM cars
        WHERE brand = ? AND model = ?
    ''', (brand, model))

    result = cursor.fetchone()
    conn.close()

```

```
if result:
    return {
        "price": result[0],
        "engine": result[1],
        "features": result[2].split(", ")
    }
else:
    return None

# Function to fetch available brands from the database
def get_available_brands():
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    cursor.execute('''
        SELECT DISTINCT brand FROM cars
    ''')

    brands = [row[0] for row in cursor.fetchall()]
    conn.close()

    return brands

# Function to fetch available models for a brand
def get_available_models(brand):
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    cursor.execute('''
        SELECT model FROM cars
        WHERE brand = ?
    ''', (brand,))

    models = [row[0] for row in cursor.fetchall()]
    conn.close()

    return models

# Chat function
def chat():
    print("Bot is running! Type 'quit' to exit")
    context = None # To keep track of the conversation context
    current_brand = None # To store the current car brand

    while True:
        inp = input("You: ")
        if inp.lower() == 'quit':
            break

        # Process input
```

```
"'Process input

inp_words = nltk.word_tokenize(inp)
inp_words = [lemmatizer.lemmatize(word.lower()) for word in inp_words]
processed_input = ' '.join(inp_words)

# Predict intent
predicted_tag = model.predict([processed_input])[0]

# Handle context
if predicted_tag == "car_info":
    context = "brand_selection"
    available_brands = get_available_brands()
    print(f"Which car brand are you interested in? Available brands: {', '.join(available_brands)}")
    continue

if context == "brand_selection":
    selected_brand = None
    available_brands = get_available_brands()
    for word in inp_words:
        if word.capitalize() in available_brands:
            selected_brand = word.capitalize()
            break

    if selected_brand:
        context = "model_selection"
        current_brand = selected_brand
        available_models = get_available_models(current_brand)
        print(f"Which {current_brand} model are you interested in? Available models: {', '.join(available_models)}")
    else:
        print(f"Sorry, I don't recognize that brand. Please choose from: {', '.join(available_brands)}")
    continue

if context == "model_selection":
    car_model = None
    available_models = get_available_models(current_brand)
    for word in inp_words:
        if word.capitalize() in available_models:
            car_model = word.capitalize()
            break

    if car_model:
        car_details = get_car_details(current_brand, car_model)
        if car_details:
            print(f"Here are the details for {current_brand} {car_model}:")
            print(f"Price: {car_details['price']}")
```

```

        print("Sorry, I couldn't find details for that car.")

        # Ask if the user wants to check another car
        print("Would you like to check another car? (yes/no)")
        context = "ask_another_car"
    else:
        print(f"Sorry, I don't recognize that {current_brand} model. Please choose from: {', '.join(available_models)}")
        continue

    if context == "ask_another_car":
        if inp.lower() in ["yes", "y"]:
            context = "brand_selection"
            available_brands = get_available_brands()
            print(f"Which car brand are you interested in? Available brands: {', '.join(available_brands)}")
        elif inp.lower() in ["no", "n"]:
            context = None
            print("Okay! Let me know if you need anything else.")
        else:
            print("Please answer with 'yes' or 'no'.")
        continue

    # Get response
    for intent in intents['intents']:
        if intent['tag'] == predicted_tag:
            responses = intent['responses']
            print(random.choice(responses))
            break

# Start chatting
chat()

```

## Key Changes

### 1. Reset After Providing Details:

- After displaying car details, the chatbot asks the user if they want to check another car.

### 2. New Context: `ask_another_car` :

- This context handles the user's response to the question: `Would you like to check another car? (yes/no)`

### 3. Reset Logic:

- If the user says `yes`, the chatbot resets to the `brand_selection` context and lists available brands again.

- If the user says **no**, the chatbot resets the context and waits for the next input.
- 

## Example Conversation

Copy

```
Bot is running! Type 'quit' to exit
You: Hi
Hello!
You: Tell me about cars
Which car brand are you interested in? Available brands: Toyota, Honda, Ford, Tesla
You: Toyota
Which Toyota model are you interested in? Available models: Corolla, Camry, Rav4
You: Camry
Here are the details for Toyota Camry:
Price: $25,000
Engine: 2.5L 4-cylinder
Features: Leather Seats, Sunroof, Android Auto
Would you like to check another car? (yes/no)
You: yes
Which car brand are you interested in? Available brands: Toyota, Honda, Ford, Tesla
You: Tesla
Which Tesla model are you interested in? Available models: Model 3, Model S
You: Model 3
Here are the details for Tesla Model 3:
Price: $45,000
Engine: Electric
Features: Autopilot, Touchscreen Display, Over-the-Air Updates
Would you like to check another car? (yes/no)
You: no
Okay! Let me know if you need anything else.
You: quit
```

---

## Benefits of This Update

### 1. Interactive and User-Friendly:

- The chatbot resets after providing details and asks if the user wants to check another car.

### 2. Flexible Conversation Flow:

- The user can continue checking cars or end the conversation as needed.

### 3. Improved Usability:

- The chatbot feels more natural and responsive to user needs.
-

## Next Steps

### 1. Add More Features:

- Allow users to compare cars, book test drives, or find dealerships.

### 2. Improve UI:

- If you're building a web-based chatbot, you can display the available brands and models as clickable buttons for better usability.

### 3. Enhance Error Handling:

- Handle cases where the database connection fails or queries return no results.

---

Let me know if you need further assistance or enhancements! 😊

my next thing is to also train the entities.



Great! Training **entities** is the next logical step to make your chatbot even more powerful. Entities are specific pieces of information extracted from user input, such as **car brands**, **models**, **colors**, or **dates**. By training entities, your chatbot can understand and extract these details dynamically, making it more flexible and capable of handling complex queries.

Here's how you can train entities and integrate them into your chatbot:

---

## Steps to Train Entities

### 1. Define Entities:

- Identify the entities you want to extract (e.g., `brand`, `model`, `color`, `price_range`).

### 2. Annotate Training Data:

- Add entity annotations to your training data (e.g., `intents.json`).

### 3. Use an NLP Library:

- Use a library like **spaCy**, **Rasa NLU**, or **NLTK** to train an entity recognition model.

### 4. Integrate Entity Recognition:

- Extract entities from user input and use them in your chatbot logic.

## Updated Chatbot Code with Entity Training

For this example, we'll use **spaCy** to train and extract entities. spaCy is a powerful NLP library that supports entity recognition.

### Step 1: Install spaCy and Download a Language Model

```
bash
```

```
Copy
```

```
pip install spacy  
python -m spacy download en_core_web_sm
```

### Step 2: Define Entities in `intents.json`

Update your `intents.json` file to include entity annotations. For example:

```
json
```

```
Copy
```

```
{  
  "intents": [  
    {  
      "tag": "car_info",  
      "patterns": [  
        "Tell me about {brand: Toyota} cars",  
        "I want to know about {brand: Honda}",  
        "What are the details of {brand: Ford} {model: Mustang}?",  
        "Tell me about {brand: Tesla} {model: Model 3}"  
      ],  
      "responses": [  
        "Which {brand} model are you interested in?",  
        "Here are the details for {brand} {model}."  
      ]  
    }  
  ]  
}
```

Here, `{brand: Toyota}` and `{model: Mustang}` are entity annotations.

### Step 3: Train Entity Recognition with spaCy

Here's how you can train a custom entity recognition model using spaCy:

```
python
```

```
Copy
```

```
import spacy
```

```

import spacy
from spacy.tokens import DocBin
from spacy.training import Example
import random

# Load a pre-trained spaCy model
nlp = spacy.load("en_core_web_sm")

# Define training data with entity annotations
TRAIN_DATA = [
    ("Tell me about Toyota cars", {"entities": [(16, 22, "BRAND")]}),
    ("I want to know about Honda", {"entities": [(20, 25, "BRAND")]}),
    ("What are the details of Ford Mustang?", {"entities": [(24, 28, "BRAND"), (29, 36, "MODEL")]}),
    ("Tell me about Tesla Model 3", {"entities": [(16, 21, "BRAND"), (22, 29, "MODEL")]})
]

# Add entity labels to the pipeline
if "ner" not in nlp.pipe_names:
    ner = nlp.add_pipe("ner")
else:
    ner = nlp.get_pipe("ner")

for _, annotations in TRAIN_DATA:
    for ent in annotations.get("entities"):
        ner.add_label(ent[2])

# Disable other pipeline components during training
other_pipes = [pipe for pipe in nlp.pipe_names if pipe != "ner"]
with nlp.disable_pipes(*other_pipes):
    optimizer = nlp.begin_training()
    for itn in range(10): # Train for 10 iterations
        random.shuffle(TRAIN_DATA)
        losses = {}
        for text, annotations in TRAIN_DATA:
            doc = nlp.make_doc(text)
            example = Example.from_dict(doc, annotations)
            nlp.update([example], drop=0.5, losses=losses)
            print(f"Iteration {itn}, Losses: {losses}")

# Save the trained model
nlp.to_disk("custom_entity_model")

```

## Step 4: Integrate Entity Recognition into the Chatbot

Here's how you can integrate the trained entity recognition model into your chatbot:

python

Copy

```

import json
import random
import spacy

# Load the custom entity recognition model
nlp = spacy.load("custom_entity_model")

# Load intents file
with open('intents.json') as file:
    intents = json.load(file)

# Function to extract entities from user input
def extract_entities(text):
    doc = nlp(text)
    entities = {}
    for ent in doc.ents:
        entities[ent.label_] = ent.text
    return entities

# Chat function
def chat():
    print("Bot is running! Type 'quit' to exit")
    while True:
        inp = input("You: ")
        if inp.lower() == 'quit':
            break

        # Extract entities
        entities = extract_entities(inp)
        brand = entities.get("BRAND")
        model = entities.get("MODEL")

        # Handle car info intent
        if brand:
            if model:
                print(f"Here are the details for {brand} {model}:")
                # Fetch and display car details from the database
            else:
                print(f"Which {brand} model are you interested in?")
                # Fetch and display available models for the brand
        else:
            print("Which car brand are you interested in?")
            # Fetch and display available brands

    # Start chatting
chat()

```

---

## Example Conversation with Entity Recognition

Copy

```
Bot is running! Type 'quit' to exit
You: Tell me about Toyota cars
Which Toyota model are you interested in?
You: What are the details of Ford Mustang?
Here are the details for Ford Mustang:
Price: $35,000
Engine: 5.0L V8
Features: Leather Seats, Premium Audio, Performance Package
You: quit
```

## Key Features of Entity Recognition

### 1. Dynamic Entity Extraction:

- The chatbot extracts `brand` and `model` entities from user input dynamically.

### 2. Flexible Queries:

- Users can ask questions in different ways (e.g., "Tell me about Toyota cars" or "What are the details of Ford Mustang?").

### 3. Scalable:

- You can add more entities (e.g., `color`, `price_range`) and train the model accordingly.

## Next Steps

### 1. Add More Entities:

- Train the model to recognize additional entities like `color`, `price_range`, or `year`.

### 2. Improve Training Data:

- Add more annotated examples to improve the accuracy of entity recognition.

### 3. Integrate with Database:

- Use extracted entities to fetch and display relevant information from the database.

### 4. Use a Pre-Trained Model:

- If you don't want to train your own model, you can use a pre-trained entity recognition model from spaCy or Rasa.

Let me know if you need further assistance or enhancements! 😊

```
{  
  "intents": [  
    {  
      "tag": "greeting",  
      "patterns": [  
        "Hi",  
        "Hello",  
        "Hey",  
        "How are you"  
      ],  
      "responses": [  
        "Hello!",  
        "Hi there!",  
        "Good to see you!"  
      ]  
    },  
    {  
      "tag": "goodbye",  
      "patterns": [  
        "Bye",  
        "See you later",  
        "Goodbye"  
      ],  
      "responses": [  
        "Goodbye!",  
        "See you later!",  
        "Take care!"  
      ]  
    },  
    {  
      "tag": "car_info",  
      "patterns": [  
        "Tell me about cars",  
        "I want to know about cars",  
        "Car details"  
      ]  
    }  
  ]  
}
```

```
],
  "responses": [
    "Which car brand are you interested in? (Toyota, Honda, Ford, Tesla)"
  ]
},
{
  "tag": "toyota_info",
  "patterns": [
    "Tell me about Toyota cars",
    "I want to know about Toyota",
    "Toyota car details"
  ],
  "responses": [
    "Which Toyota model are you interested in?"
  ]
},
{
  "tag": "honda_info",
  "patterns": [
    "Tell me about Honda cars",
    "I want to know about Honda",
    "Honda car details"
  ],
  "responses": [
    "Which Honda model are you interested in?"
  ]
},
{
  "tag": "ford_info",
  "patterns": [
    "Tell me about Ford cars",
    "I want to know about Ford",
    "Ford car details"
  ],
  "responses": [
    "Which Ford model are you interested in?"
  ]
}
```

```
        ],
    },
    {
        "tag": "tesla_info",
        "patterns": [
            "Tell me about Tesla cars",
            "I want to know about Tesla",
            "Tesla car details"
        ],
        "responses": [
            "Which Tesla model are you interested in?"
        ]
    }
]
```

}This is the intent file. modeify the above code accorning to it



To modify the code according to your `intents.json` file, we need to integrate the intents and responses into the chatbot logic. The updated code will use the intents and responses from your `intents.json` file while maintaining the entity recognition and database integration.

Here's the updated code:

---

#### Updated Chatbot Code with Your `intents.json`

```
python                                         Copy

import json
import random
import nltk
import sqlite3
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline

# Download NLTK data
nltk.download('punkt')
nltk.download('wordnet')
```

```

# Initialize lemmatizer
lemmatizer = WordNetLemmatizer()

# Load intents file
with open('intents.json') as file:
    intents = json.load(file)

# Preprocess data
words = []
classes = []
documents = []
ignore_chars = ['?', '!', '.', ',']

for intent in intents['intents']:
    for pattern in intent['patterns']:
        # Tokenize words
        word_list = nltk.word_tokenize(pattern)
        words.extend(word_list)
        documents.append(' '.join(word_list), intent['tag']))
        if intent['tag'] not in classes:
            classes.append(intent['tag'])

# Lemmatize and clean words
words = [lemmatizer.lemmatize(word.lower()) for word in words if word not in ignore_cha
rs]
words = sorted(list(set(words)))
classes = sorted(list(set(classes)))

# Prepare training data
X_train = [] # Patterns
y_train = [] # Intents (tags)

for doc in documents:
    X_train.append(doc[0]) # Pattern
    y_train.append(doc[1]) # Tag

# Create a pipeline with TF-IDF and SVM
model = make_pipeline(TfidfVectorizer(), SVC(kernel='linear', probability=True))

# Train the model
model.fit(X_train, y_train)

# Function to fetch car details from the database
def get_car_details(brand, model):
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    cursor.execute('''
        SELECT price, engine, features FROM cars
        WHERE brand = ? AND model = ?
    ''')

```

```
    '''', (brand, model))

    result = cursor.fetchone()
    conn.close()

    if result:
        return {
            "price": result[0],
            "engine": result[1],
            "features": result[2].split(", ")
        }
    else:
        return None

# Function to fetch available brands from the database
def get_available_brands():
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    cursor.execute('''
    SELECT DISTINCT brand FROM cars
    ''')

    brands = [row[0] for row in cursor.fetchall()]
    conn.close()

    return brands

# Function to fetch available models for a brand
def get_available_models(brand):
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    cursor.execute('''
    SELECT model FROM cars
    WHERE brand = ?
    ''', (brand,))

    models = [row[0] for row in cursor.fetchall()]
    conn.close()

    return models

# Chat function
def chat():
    print("Bot is running! Type 'quit' to exit")
    context = None # To keep track of the conversation context
    current_brand = None # To store the current car brand

    while True:
```

```
inp = input("You: ")
if inp.lower() == 'quit':
    break

# Process input
inp_words = nltk.word_tokenize(inp)
inp_words = [lemmatizer.lemmatize(word.lower()) for word in inp_words]
processed_input = ' '.join(inp_words)

# Predict intent
predicted_tag = model.predict([processed_input])[0]

# Get response based on predicted intent
for intent in intents['intents']:
    if intent['tag'] == predicted_tag:
        responses = intent['responses']
        response = random.choice(responses)
        break

# Handle car_info intent
if predicted_tag == "car_info":
    context = "brand_selection"
    available_brands = get_available_brands()
    print(response) # "Which car brand are you interested in? (Toyota, Honda,
Ford, Tesla)"
    continue

# Handle brand-specific intents (e.g., toyota_info, honda_info)
if predicted_tag.endswith("_info"):
    context = "model_selection"
    current_brand = predicted_tag.replace("_info", "").capitalize()
    available_models = get_available_models(current_brand)
    print(response) # "Which Toyota model are you interested in?"
    continue

# Handle model selection
if context == "model_selection":
    car_model = None
    available_models = get_available_models(current_brand)
    for word in inp_words:
        if word.capitalize() in available_models:
            car_model = word.capitalize()
            break

    if car_model:
        car_details = get_car_details(current_brand, car_model)
        if car_details:
            print(f"Here are the details for {current_brand} {car_model}:")
            print(f"Price: {car_details['price']}")
            print(f"Engine: {car_details['engine']}")
```

```

        print(f"Features: {', '.join(car_details['features'])}")
    else:
        print("Sorry, I couldn't find details for that car.")

    # Ask if the user wants to check another car
    print("Would you like to check another car? (yes/no)")
    context = "ask_another_car"
else:
    print(f"Sorry, I don't recognize that {current_brand} model. Please choose from: {', '.join(available_models)}")
    continue

# Handle ask_another_car context
if context == "ask_another_car":
    if inp.lower() in ["yes", "y"]:
        context = "brand_selection"
        available_brands = get_available_brands()
        print(f"Which car brand are you interested in? Available brands: {', '.join(available_brands)}")
    elif inp.lower() in ["no", "n"]:
        context = None
        print("Okay! Let me know if you need anything else.")
    else:
        print("Please answer with 'yes' or 'no'.")
    continue

# Default response
print(response)

# Start chatting
chat()

```

---

## Key Changes

### 1. Dynamic Responses:

- The chatbot uses the `responses` from your `intents.json` file to respond to user queries.

### 2. Brand-Specific Intents:

- The chatbot handles brand-specific intents (e.g., `toyota_info`, `honda_info`) and asks for the model.

### 3. Context Handling:

- The chatbot maintains context for brand selection, model selection, and asking if the user wants to check another car.

### 4. Database Integration:

- The chatbot fetches car details from the SQLite database based on the selected brand and model.
- 

## Example Conversation

Copy

```
Bot is running! Type 'quit' to exit
You: Hi
Hello!
You: Tell me about cars
Which car brand are you interested in? (Toyota, Honda, Ford, Tesla)
You: Toyota
Which Toyota model are you interested in?
You: Camry
Here are the details for Toyota Camry:
Price: $25,000
Engine: 2.5L 4-cylinder
Features: Leather Seats, Sunroof, Android Auto
Would you like to check another car? (yes/no)
You: yes
Which car brand are you interested in? Available brands: Toyota, Honda, Ford, Tesla
You: Tesla
Which Tesla model are you interested in?
You: Model 3
Here are the details for Tesla Model 3:
Price: $45,000
Engine: Electric
Features: Autopilot, Touchscreen Display, Over-the-Air Updates
Would you like to check another car? (yes/no)
You: no
Okay! Let me know if you need anything else.
You: quit
```

---

## Benefits of This Update

### 1. Dynamic and Scalable:

- The chatbot uses your `intents.json` file to handle intents and responses dynamically.

### 2. User-Friendly:

- The chatbot guides the user step-by-step, listing available brands and models.

### 3. Flexible Conversation Flow:

- The chatbot resets after providing details and asks if the user wants to check

another car.

---

## Next Steps

### 1. Add More Intents:

- Add more intents and responses to handle additional queries (e.g., comparing cars, booking test drives).

### 2. Improve Entity Recognition:

- Train the chatbot to recognize entities like `color`, `price_range`, or `year`.

### 3. Enhance UI:

- If you're building a web-based chatbot, display available brands and models as clickable buttons.
- 

Let me know if you need further assistance or enhancements! 😊

i want to add the entity training to recognise entity for the baove json file



To add **entity recognition** to your chatbot, we need to train a model to recognize entities like `brand` and `model` from user input. We'll use **spaCy**, a powerful NLP library, to train a custom entity recognition model. This will allow the chatbot to dynamically extract entities like `Toyota` (brand) and `Camry` (model) from user queries.

Here's how you can integrate entity training and recognition into your chatbot:

---

### Step 1: Install spaCy and Download a Language Model

```
bash
```

Copy

```
pip install spacy  
python -m spacy download en_core_web_sm
```

### Step 2: Prepare Training Data for Entity Recognition

We need to annotate the training data with entities. For example:

```
python
```

Copy

```
TRAIN_DATA = [
    ("Tell me about Toyota cars", {"entities": [(16, 22, "BRAND")]}),
    ("I want to know about Honda", {"entities": [(20, 25, "BRAND")]}),
    ("What are the details of Ford Mustang?", {"entities": [(24, 28, "BRAND"), (29, 36, "MODEL")]}),
    ("Tell me about Tesla Model 3", {"entities": [(16, 21, "BRAND"), (22, 29, "MODEL")]})
]
```

Here, `BRAND` and `MODEL` are the entity labels.

## Step 3: Train the Entity Recognition Model

Here's the code to train a custom entity recognition model using spaCy:

```
python Copy
import spacy
from spacy.tokens import DocBin
from spacy.training import Example
import random

# Load a pre-trained spaCy model
nlp = spacy.load("en_core_web_sm")

# Define training data with entity annotations
TRAIN_DATA = [
    ("Tell me about Toyota cars", {"entities": [(16, 22, "BRAND")]}),
    ("I want to know about Honda", {"entities": [(20, 25, "BRAND")]}),
    ("What are the details of Ford Mustang?", {"entities": [(24, 28, "BRAND"), (29, 36, "MODEL")]}),
    ("Tell me about Tesla Model 3", {"entities": [(16, 21, "BRAND"), (22, 29, "MODEL")]})
]

# Add entity labels to the pipeline
if "ner" not in nlp.pipe_names:
    ner = nlp.add_pipe("ner")
else:
    ner = nlp.get_pipe("ner")

for _, annotations in TRAIN_DATA:
    for ent in annotations.get("entities"):
        ner.add_label(ent[2])

# Disable other pipeline components during training
other_pipes = [pipe for pipe in nlp.pipe_names if pipe != "ner"]
with nlp.disable_pipes(*other_pipes):
```

```
optimizer = nlp.begin_training()
for itn in range(10): # Train for 10 iterations
    random.shuffle(TRAIN_DATA)
    losses = {}
    for text, annotations in TRAIN_DATA:
        doc = nlp.make_doc(text)
        example = Example.from_dict(doc, annotations)
        nlp.update([example], drop=0.5, losses=losses)
    print(f"Iteration {itn}, Losses: {losses}")

# Save the trained model
nlp.to_disk("custom_entity_model")
```

## Step 4: Integrate Entity Recognition into the Chatbot

Here's the updated chatbot code with entity recognition:

```
python
import json
import random
import nltk
import sqlite3
import spacy
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline

# Download NLTK data
nltk.download('punkt')
nltk.download('wordnet')

# Initialize lemmatizer
lemmatizer = WordNetLemmatizer()

# Load intents file
with open('intents.json') as file:
    intents = json.load(file)

# Load the custom entity recognition model
nlp = spacy.load("custom_entity_model")

# Preprocess data
words = []
classes = []
documents = []
ignore_chars = ['?', '!', '.', ',']
```

Copy

```
for intent in intents['intents']:
    for pattern in intent['patterns']:
        # Tokenize words
        word_list = nltk.word_tokenize(pattern)
        words.extend(word_list)
        documents.append(( ' '.join(word_list), intent['tag']))
        if intent['tag'] not in classes:
            classes.append(intent['tag'])

# Lemmatize and clean words
words = [lemmatizer.lemmatize(word.lower()) for word in words if word not in ignore_chars]
words = sorted(list(set(words)))
classes = sorted(list(set(classes)))

# Prepare training data
X_train = [] # Patterns
y_train = [] # Intents (tags)

for doc in documents:
    X_train.append(doc[0]) # Pattern
    y_train.append(doc[1]) # Tag

# Create a pipeline with TF-IDF and SVM
model = make_pipeline(TfidfVectorizer(), SVC(kernel='linear', probability=True))

# Train the model
model.fit(X_train, y_train)

# Function to fetch car details from the database
def get_car_details(brand, model):
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    cursor.execute('''
    SELECT price, engine, features FROM cars
    WHERE brand = ? AND model = ?
    ''', (brand, model))

    result = cursor.fetchone()
    conn.close()

    if result:
        return {
            "price": result[0],
            "engine": result[1],
            "features": result[2].split(", ")
        }
    else:
        return None
```

```

# Function to extract entities from user input
def extract_entities(text):
    doc = nlp(text)
    entities = {}
    for ent in doc.ents:
        entities[ent.label_] = ent.text
    return entities

# Chat function
def chat():
    print("Bot is running! Type 'quit' to exit")
    context = None # To keep track of the conversation context
    current_brand = None # To store the current car brand

    while True:
        inp = input("You: ")
        if inp.lower() == 'quit':
            break

        # Extract entities
        entities = extract_entities(inp)
        brand = entities.get("BRAND")
        model = entities.get("MODEL")

        # Predict intent
        processed_input = ' '.join([lemmatizer.lemmatize(word.lower()) for word in nltk.word_tokenize(inp)])
        predicted_tag = model.predict([processed_input])[0]

        # Get response based on predicted intent
        for intent in intents['intents']:
            if intent['tag'] == predicted_tag:
                responses = intent['responses']
                response = random.choice(responses)
                break

        # Handle car_info intent
        if predicted_tag == "car_info":
            if brand:
                context = "model_selection"
                current_brand = brand
                available_models = get_available_models(current_brand)
                print(f"Which {current_brand} model are you interested in? Available models: {', '.join(available_models)}")
            else:
                context = "brand_selection"
                available_brands = get_available_brands()
                print(response) # "Which car brand are you interested in? (Toyota, Honda, Ford, Tesla)"


```

```
        continue

    # Handle brand-specific intents (e.g., toyota_info, honda_info)
    if predicted_tag.endswith("_info"):
        context = "model_selection"
        current_brand = predicted_tag.replace("_info", "").capitalize()
        available_models = get_available_models(current_brand)
        print(response) # "Which Toyota model are you interested in?"
        continue

    # Handle model selection
    if context == "model_selection":
        if model:
            car_details = get_car_details(current_brand, model)
            if car_details:
                print(f"Here are the details for {current_brand} {model}:")
                print(f"Price: {car_details['price']} ")
                print(f"Engine: {car_details['engine']} ")
                print(f"Features: {', '.join(car_details['features'])} ")
            else:
                print("Sorry, I couldn't find details for that car.")

        # Ask if the user wants to check another car
        print("Would you like to check another car? (yes/no)")
        context = "ask_another_car"
    else:
        available_models = get_available_models(current_brand)
        print(f"Sorry, I don't recognize that {current_brand} model. Please choose from: {', '.join(available_models)}")
        continue

    # Handle ask_another_car context
    if context == "ask_another_car":
        if inp.lower() in ["yes", "y"]:
            context = "brand_selection"
            available_brands = get_available_brands()
            print(f"Which car brand are you interested in? Available brands: {', '.join(available_brands)}")
        elif inp.lower() in ["no", "n"]:
            context = None
            print("Okay! Let me know if you need anything else.")
        else:
            print("Please answer with 'yes' or 'no'.")
        continue

    # Default response
    print(response)

# Start chatting
chat()
```

---

## Key Changes

### 1. Entity Recognition:

- The `extract_entities` function uses the trained spaCy model to extract `BRAND` and `MODEL` entities from user input.

### 2. Dynamic Entity Handling:

- If the user provides a brand or model in their query, the chatbot skips the intermediate steps and directly fetches the details.

### 3. Context Handling:

- The chatbot maintains context for brand selection, model selection, and asking if the user wants to check another car.

---

## Example Conversation with Entity Recognition

```
Copy

Bot is running! Type 'quit' to exit
You: Hi
Hello!
You: Tell me about Toyota Camry
Here are the details for Toyota Camry:
Price: $25,000
Engine: 2.5L 4-cylinder
Features: Leather Seats, Sunroof, Android Auto
Would you like to check another car? (yes/no)
You: yes
Which car brand are you interested in? Available brands: Toyota, Honda, Ford, Tesla
You: Tesla Model 3
Here are the details for Tesla Model 3:
Price: $45,000
Engine: Electric
Features: Autopilot, Touchscreen Display, Over-the-Air Updates
Would you like to check another car? (yes/no)
You: no
Okay! Let me know if you need anything else.
You: quit
```

---

## Benefits of Entity Recognition

### 1. Dynamic Entity Extraction:

- The chatbot extracts `brand` and `model` entities from user input dynamically.

## 2. Flexible Queries:

- Users can ask questions in different ways (e.g., "Tell me about Toyota Camry" or "What are the details of Ford Mustang?").

## 3. Improved Usability:

- The chatbot feels more natural and responsive to user needs.
- 

## Next Steps

### 1. Add More Entities:

- Train the model to recognize additional entities like `color`, `price_range`, or `year`.

### 2. Improve Training Data:

- Add more annotated examples to improve the accuracy of entity recognition.

### 3. Enhance UI:

- If you're building a web-based chatbot, display available brands and models as clickable buttons.
- 

Let me know if you need further assistance or enhancements! 😊

in the entity recognition model you have used TRAIN\_DATA = [  
("Tell me about Toyota cars", {"entities": [(16, 22, "BRAND")]}),  
("I want to know about Honda", {"entities": [(20, 25, "BRAND")]}),  
("What are the details of Ford Mustang?", {"entities": [(24, 28, "BRAND"), (29, 36, "MODEL")]}),  
("Tell me about Tesla Model 3", {"entities": [(16, 21, "BRAND"), (22, 29, "MODEL")]}]  
] that much of training data. can you use my intent file i have given to create atrian  
data fro it



The server is busy. Please try again later.

< 2 / 2 >

n the entity recognition model you have used TRAIN\_DATA = [  
("Tell me about Toyota cars", {"entities": [(16, 22, "BRAND")]}),  
("I want to know about Honda", {"entities": [(20, 25, "BRAND")]}),

("What are the details of Ford Mustang?", {"entities": [(24, 28, "BRAND"), (29, 36, "MODEL")]}),  
("Tell me about Tesla Model 3", {"entities": [(16, 21, "BRAND"), (22, 29, "MODEL")]}))  
] that much of training data. can you use my intent file i have given to create training  
data from it

Yes, we can use your `intents.json` file to create training data for entity recognition. The idea is to extract patterns from your `intents.json` file and annotate them with entities like `BRAND` and `MODEL`. This way, the entity recognition model can learn to identify these entities from user input.

Here's how we can generate training data for entity recognition from your `intents.json` file:

## Step 1: Extract Patterns and Annotate Entities

We'll parse your `intents.json` file and create training data for entity recognition. For example:

- If a pattern contains a brand (e.g., "Tell me about Toyota cars"), we'll annotate `Toyota` as a `BRAND`.
- If a pattern contains a model (e.g., "What are the details of Ford Mustang?"), we'll annotate `Ford` as a `BRAND` and `Mustang` as a `MODEL`.

## Step 2: Generate Training Data

Here's the Python script to generate training data for entity recognition from your `intents.json` file:

```
python
import json

# Load intents file
with open('intents.json') as file:
    intents = json.load(file)

# Function to generate training data for entity recognition
def generate_training_data(intents):
    training_data = []
```

```

for intent in intents['intents']:
    for pattern in intent['patterns']:
        # Initialize entities list
        entities = []

        # Check for brand and model in the pattern
        if "Toyota" in pattern:
            start = pattern.find("Toyota")
            entities.append((start, start + len("Toyota"), "BRAND"))
        if "Honda" in pattern:
            start = pattern.find("Honda")
            entities.append((start, start + len("Honda"), "BRAND"))
        if "Ford" in pattern:
            start = pattern.find("Ford")
            entities.append((start, start + len("Ford"), "BRAND"))
        if "Tesla" in pattern:
            start = pattern.find("Tesla")
            entities.append((start, start + len("Tesla"), "BRAND"))
        if "Mustang" in pattern:
            start = pattern.find("Mustang")
            entities.append((start, start + len("Mustang"), "MODEL"))
        if "Model 3" in pattern:
            start = pattern.find("Model 3")
            entities.append((start, start + len("Model 3"), "MODEL"))

        # Add to training data if entities are found
        if entities:
            training_data.append((pattern, {"entities": entities}))

return training_data

# Generate training data
TRAIN_DATA = generate_training_data(intents)

# Print training data
for data in TRAIN_DATA:
    print(data)

```

## Example Output

The script will generate training data like this:

[Copy](#)

```

('Tell me about Toyota cars', {'entities': [(16, 22, 'BRAND')]})
('I want to know about Honda', {'entities': [(20, 25, 'BRAND')]})
('What are the details of Ford Mustang?', {'entities': [(24, 28, 'BRAND'), (29, 36, 'MODEL')]})

```

```
    'Tell me about Tesla Model 3', {'entities': [(16, 21, 'BRAND'), (22, 29, 'MODEL')]}))
```

## Step 3: Train the Entity Recognition Model

Now that we have the training data, we can train the entity recognition model using spaCy. Here's the updated code to train the model:

```
python
```

```
Copy
```

```
import spacy
from spacy.tokens import DocBin
from spacy.training import Example
import random

# Load a pre-trained spaCy model
nlp = spacy.load("en_core_web_sm")

# Add entity labels to the pipeline
if "ner" not in nlp.pipe_names:
    ner = nlp.add_pipe("ner")
else:
    ner = nlp.get_pipe("ner")

for _, annotations in TRAIN_DATA:
    for ent in annotations.get("entities"):
        ner.add_label(ent[2])

# Disable other pipeline components during training
other_pipes = [pipe for pipe in nlp.pipe_names if pipe != "ner"]
with nlp.disable_pipes(*other_pipes):
    optimizer = nlp.begin_training()
    for itn in range(10): # Train for 10 iterations
        random.shuffle(TRAIN_DATA)
        losses = {}
        for text, annotations in TRAIN_DATA:
            doc = nlp.make_doc(text)
            example = Example.from_dict(doc, annotations)
            nlp.update([example], drop=0.5, losses=losses)
            print(f"Iteration {itn}, Losses: {losses}")

# Save the trained model
nlp.to_disk("custom_entity_model")
```

## Step 4: Integrate Entity Recognition into the Chatbot

Here's the updated chatbot code with entity recognition:

python

Copy

```
import json
import random
import nltk
import sqlite3
import spacy
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline

# Download NLTK data
nltk.download('punkt')
nltk.download('wordnet')

# Initialize lemmatizer
lemmatizer = WordNetLemmatizer()

# Load intents file
with open('intents.json') as file:
    intents = json.load(file)

# Load the custom entity recognition model
nlp = spacy.load("custom_entity_model")

# Preprocess data
words = []
classes = []
documents = []
ignore_chars = ['?', '!', '.', ',']

for intent in intents['intents']:
    for pattern in intent['patterns']:
        # Tokenize words
        word_list = nltk.word_tokenize(pattern)
        words.extend(word_list)
        documents.append(' '.join(word_list), intent['tag']))
        if intent['tag'] not in classes:
            classes.append(intent['tag'])

# Lemmatize and clean words
words = [lemmatizer.lemmatize(word.lower()) for word in words if word not in ignore_chars]
words = sorted(list(set(words)))
classes = sorted(list(set(classes)))

# Prepare training data
X_train = [] # Patterns
y_train = [] # Intents (tags)
```

```

for doc in documents:
    X_train.append(doc[0]) # Pattern
    y_train.append(doc[1]) # Tag

# Create a pipeline with TF-IDF and SVM
model = make_pipeline(TfidfVectorizer(), SVC(kernel='linear', probability=True))

# Train the model
model.fit(X_train, y_train)

# Function to fetch car details from the database
def get_car_details(brand, model):
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    cursor.execute('''
        SELECT price, engine, features FROM cars
        WHERE brand = ? AND model = ?
    ''', (brand, model))

    result = cursor.fetchone()
    conn.close()

    if result:
        return {
            "price": result[0],
            "engine": result[1],
            "features": result[2].split(", ")
        }
    else:
        return None

# Function to extract entities from user input
def extract_entities(text):
    doc = nlp(text)
    entities = {}
    for ent in doc.ents:
        entities[ent.label_] = ent.text
    return entities

# Chat function
def chat():
    print("Bot is running! Type 'quit' to exit")
    context = None # To keep track of the conversation context
    current_brand = None # To store the current car brand

    while True:
        inp = input("You: ")
        if inp.lower() == 'quit':

```

```

        break

    # Extract entities
    entities = extract_entities(inp)
    brand = entities.get("BRAND")
    model = entities.get("MODEL")

    # Predict intent
    processed_input = ' '.join([lemmatizer.lemmatize(word.lower()) for word in nlt
k.word_tokenize(inp)])
    predicted_tag = model.predict([processed_input])[0]

    # Get response based on predicted intent
    for intent in intents['intents']:
        if intent['tag'] == predicted_tag:
            responses = intent['responses']
            response = random.choice(responses)
            break

    # Handle car_info intent
    if predicted_tag == "car_info":
        if brand:
            context = "model_selection"
            current_brand = brand
            available_models = get_available_models(current_brand)
            print(f"Which {current_brand} model are you interested in? Available mo
dels: {', '.join(available_models)}")
        else:
            context = "brand_selection"
            available_brands = get_available_brands()
            print(response) # "Which car brand are you interested in? (Toyota, Hon
da, Ford, Tesla)"
            continue

    # Handle brand-specific intents (e.g., toyota_info, honda_info)
    if predicted_tag.endswith("_info"):
        context = "model_selection"
        current_brand = predicted_tag.replace("_info", "").capitalize()
        available_models = get_available_models(current_brand)
        print(response) # "Which Toyota model are you interested in?"
        continue

    # Handle model selection
    if context == "model_selection":
        if model:
            car_details = get_car_details(current_brand, model)
            if car_details:
                print(f"Here are the details for {current_brand} {model}:")
                print(f"Price: {car_details['price']}")
                print(f"Engine: {car_details['engine']}")
```

```

        print(f"Features: {', '.join(car_details['features'])})")
    else:
        print("Sorry, I couldn't find details for that car.")

    # Ask if the user wants to check another car
    print("Would you like to check another car? (yes/no)")
    context = "ask_another_car"
else:
    available_models = get_available_models(current_brand)
    print(f"Sorry, I don't recognize that {current_brand} model. Please choose from: {', '.join(available_models)}")
    continue

    # Handle ask_another_car context
    if context == "ask_another_car":
        if inp.lower() in ["yes", "y"]:
            context = "brand_selection"
            available_brands = get_available_brands()
            print(f"Which car brand are you interested in? Available brands: {', '.join(available_brands)}")
        elif inp.lower() in ["no", "n"]:
            context = None
            print("Okay! Let me know if you need anything else.")
        else:
            print("Please answer with 'yes' or 'no'.")
    continue

    # Default response
    print(response)

# Start chatting
chat()

```

---

## Key Changes

### 1. Training Data Generation:

- The `generate_training_data` function creates training data for entity recognition from your `intents.json` file.

### 2. Entity Recognition Integration:

- The chatbot uses the trained spaCy model to extract `BRAND` and `MODEL` entities from user input.

### 3. Dynamic Entity Handling:

- If the user provides a brand or model in their query, the chatbot skips the intermediate steps and directly fetches the details.

## Example Conversation with Entity Recognition

Copy

```
Bot is running! Type 'quit' to exit
You: Hi
Hello!
You: Tell me about Toyota Camry
Here are the details for Toyota Camry:
Price: $25,000
Engine: 2.5L 4-cylinder
Features: Leather Seats, Sunroof, Android Auto
Would you like to check another car? (yes/no)
You: yes
Which car brand are you interested in? Available brands: Toyota, Honda, Ford, Tesla
You: Tesla Model 3
Here are the details for Tesla Model 3:
Price: $45,000
Engine: Electric
Features: Autopilot, Touchscreen Display, Over-the-Air Updates
Would you like to check another car? (yes/no)
You: no
Okay! Let me know if you need anything else.
You: quit
```

## Benefits of Entity Recognition

### 1. Dynamic Entity Extraction:

- The chatbot extracts `brand` and `model` entities from user input dynamically.

### 2. Flexible Queries:

- Users can ask questions in different ways (e.g., "Tell me about Toyota Camry" or "What are the details of Ford Mustang?").

### 3. Improved Usability:

- The chatbot feels more natural and responsive to user needs.

## Next Steps

### 1. Add More Entities:

- Train the model to recognize additional entities like `color`, `price_range`, or `year`.

### 2. Improve Training Data:

- Add more annotated examples to improve the accuracy of entity recognition.

### 3. Enhance UI:

- If you're building a web-based chatbot, display available brands and models as clickable buttons.

---

Let me know if you need further assistance or enhancements! 😊

```
{  
  "intents": [  
    {  
      "tag": "greeting",  
      "patterns": [  
        "Hi",  
        "Hello",  
        "Hey",  
        "How are you"  
      ],  
      "responses": [  
        "Hello!",  
        "Hi there!",  
        "Good to see you!"  
      ]  
    },  
    {  
      "tag": "goodbye",  
      "patterns": [  
        "Bye",  
        "See you later",  
        "Goodbye"  
      ],  
      "responses": [  
        "Goodbye!",  
        "See you later!",  
        "Take care!"  
      ]  
    }]
```

```
        },
        {
            "tag": "car_info",
            "patterns": [
                "Tell me about cars",
                "I want to know about cars",
                "Car details"
            ],
            "responses": [
                "Which car brand are you interested in? (Toyota, Honda, Ford, Tesla)"
            ]
        },
        {
            "tag": "toyota_info",
            "patterns": [
                "Tell me about Toyota cars",
                "I want to know about Toyota",
                "Toyota car details"
            ],
            "responses": [
                "Which Toyota model are you interested in?"
            ]
        },
        {
            "tag": "honda_info",
            "patterns": [
                "Tell me about Honda cars",
                "I want to know about Honda",
                "Honda car details"
            ],
            "responses": [
                "Which Honda model are you interested in?"
            ]
        },
        {
            "tag": "ford_info",
            "patterns": [
                "Tell me about Ford cars",
                "I want to know about Ford"
            ],
            "responses": [
                "Which Ford model are you interested in?"
            ]
        }
    ]
}
```

```
patterns": [
    "Tell me about Ford cars",
    "I want to know about Ford",
    "Ford car details"
],
"responses": [
    "Which Ford model are you interested in?"
]
},
{
    "tag": "tesla_info",
    "patterns": [
        "Tell me about Tesla cars",
        "I want to know about Tesla",
        "Tesla car details"
    ],
    "responses": [
        "Which Tesla model are you interested in?"
    ]
}
]
}

} generate for this
```



i have database from whcih i need to create entity recognithing script



The server is busy. Please try again later.

< 4 / 4 >

i have database from whcih i need to create entity recognithing script



If you have a **database** that contains car brands, models, and other relevant information,

we can use this data to create a **custom entity recognition script**. The idea is to extract patterns from the database and annotate them with entities like **BRAND** and **MODEL**. This will allow the chatbot to dynamically recognize entities from user input.

Here's how you can create an entity recognition script using data from your database:

---

## Step 1: Connect to the Database and Fetch Data

First, connect to your database and fetch the relevant data (e.g., car brands and models). For this example, I'll assume you're using a **SQLite database** with a table named **cars** that has columns **brand** and **model**.

```
python                                         Copy

import sqlite3

# Connect to the SQLite database
conn = sqlite3.connect('car_database.db')
cursor = conn.cursor()

# Fetch all car brands and models
cursor.execute('SELECT brand, model FROM cars')
car_data = cursor.fetchall()

# Close the connection
conn.close()

# Print the fetched data
for brand, model in car_data:
    print(f"Brand: {brand}, Model: {model}")
```

---

## Step 2: Generate Training Data for Entity Recognition

Using the data fetched from the database, we can generate training data for entity recognition. For example:

- If a user says, "Tell me about Toyota Camry," we'll annotate **Toyota** as **BRAND** and **Camry** as **MODEL**.

Here's the script to generate training data:

```
python                                         Copy

# Function to generate training data for entity recognition
def generate_training_data(car_data):
```

```

def generate_training_data(car_data):
    training_data = []

    for brand, model in car_data:
        # Create patterns and annotate entities
        pattern1 = f"Tell me about {brand} {model}"
        entities1 = [
            (pattern1.find(brand), pattern1.find(brand) + len(brand), "BRAND"),
            (pattern1.find(model), pattern1.find(model) + len(model), "MODEL")
        ]
        training_data.append((pattern1, {"entities": entities1}))

        pattern2 = f"What are the details of {brand} {model}?"
        entities2 = [
            (pattern2.find(brand), pattern2.find(brand) + len(brand), "BRAND"),
            (pattern2.find(model), pattern2.find(model) + len(model), "MODEL")
        ]
        training_data.append((pattern2, {"entities": entities2}))

    return training_data

# Generate training data
TRAIN_DATA = generate_training_data(car_data)

# Print training data
for data in TRAIN_DATA:
    print(data)

```

## Example Output

The script will generate training data like this:

```

Copy

('Tell me about Toyota Camry', {'entities': [(16, 22, 'BRAND'), (23, 28, 'MODEL')]})
('What are the details of Toyota Camry?', {'entities': [(24, 30, 'BRAND'), (31, 36, 'MODEL')]})
('Tell me about Honda Civic', {'entities': [(16, 21, 'BRAND'), (22, 27, 'MODEL')]})
('What are the details of Honda Civic?', {'entities': [(24, 29, 'BRAND'), (30, 35, 'MODEL')]})

```

## Step 3: Train the Entity Recognition Model

Now that we have the training data, we can train the entity recognition model using **spaCy**. Here's the code to train the model:

python

Copy

```

import spacy
from spacy.tokens import DocBin
from spacy.training import Example
import random

# Load a pre-trained spaCy model
nlp = spacy.load("en_core_web_sm")

# Add entity labels to the pipeline
if "ner" not in nlp.pipe_names:
    ner = nlp.add_pipe("ner")
else:
    ner = nlp.get_pipe("ner")

for _, annotations in TRAIN_DATA:
    for ent in annotations.get("entities"):
        ner.add_label(ent[2])

# Disable other pipeline components during training
other_pipes = [pipe for pipe in nlp.pipe_names if pipe != "ner"]
with nlp.disable_pipes(*other_pipes):
    optimizer = nlp.begin_training()
    for itn in range(10): # Train for 10 iterations
        random.shuffle(TRAIN_DATA)
        losses = {}
        for text, annotations in TRAIN_DATA:
            doc = nlp.make_doc(text)
            example = Example.from_dict(doc, annotations)
            nlp.update([example], drop=0.5, losses=losses)
            print(f"Iteration {itn}, Losses: {losses}")

# Save the trained model
nlp.to_disk("custom_entity_model")

```

## Step 4: Integrate Entity Recognition into the Chatbot

Here's how you can integrate the trained entity recognition model into your chatbot:

python

Copy

```

import json
import random
import nltk
import sqlite3
import spacy
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline

```

```
# Download NLTK data
nltk.download('punkt')
nltk.download('wordnet')

# Initialize lemmatizer
lemmatizer = WordNetLemmatizer()

# Load intents file
with open('intents.json') as file:
    intents = json.load(file)

# Load the custom entity recognition model
nlp = spacy.load("custom_entity_model")

# Preprocess data
words = []
classes = []
documents = []
ignore_chars = ['?', '!', '.', ',']

for intent in intents['intents']:
    for pattern in intent['patterns']:
        # Tokenize words
        word_list = nltk.word_tokenize(pattern)
        words.extend(word_list)
        documents.append(' '.join(word_list), intent['tag']))
        if intent['tag'] not in classes:
            classes.append(intent['tag'])

# Lemmatize and clean words
words = [lemmatizer.lemmatize(word.lower()) for word in words if word not in ignore_chars]
words = sorted(list(set(words)))
classes = sorted(list(set(classes)))

# Prepare training data
X_train = [] # Patterns
y_train = [] # Intents (tags)

for doc in documents:
    X_train.append(doc[0]) # Pattern
    y_train.append(doc[1]) # Tag

# Create a pipeline with TF-IDF and SVM
model = make_pipeline(TfidfVectorizer(), SVC(kernel='linear', probability=True))

# Train the model
model.fit(X_train, y_train)
```

```

# Function to fetch car details from the database
def get_car_details(brand, model):
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    cursor.execute('''
        SELECT price, engine, features FROM cars
        WHERE brand = ? AND model = ?
    ''', (brand, model))

    result = cursor.fetchone()
    conn.close()

    if result:
        return {
            "price": result[0],
            "engine": result[1],
            "features": result[2].split(", ")
        }
    else:
        return None

# Function to extract entities from user input
def extract_entities(text):
    doc = nlp(text)
    entities = {}
    for ent in doc.ents:
        entities[ent.label_] = ent.text
    return entities

# Chat function
def chat():
    print("Bot is running! Type 'quit' to exit")
    context = None # To keep track of the conversation context
    current_brand = None # To store the current car brand

    while True:
        inp = input("You: ")
        if inp.lower() == 'quit':
            break

        # Extract entities
        entities = extract_entities(inp)
        brand = entities.get("BRAND")
        model = entities.get("MODEL")

        # Predict intent
        processed_input = ' '.join([lemmatizer.lemmatize(word.lower()) for word in nlt
k.word_tokenize(inp)])
        predicted_tag = model.predict([processed_input])[0]

```

```

# Get response based on predicted intent
for intent in intents['intents']:
    if intent['tag'] == predicted_tag:
        responses = intent['responses']
        response = random.choice(responses)
        break

# Handle car_info intent
if predicted_tag == "car_info":
    if brand:
        context = "model_selection"
        current_brand = brand
        available_models = get_available_models(current_brand)
        print(f"Which {current_brand} model are you interested in? Available mo
dels: {' , '.join(available_models)}")
    else:
        context = "brand_selection"
        available_brands = get_available_brands()
        print(response) # "Which car brand are you interested in? (Toyota, Hon
da, Ford, Tesla)"
        continue

# Handle brand-specific intents (e.g., toyota_info, honda_info)
if predicted_tag.endswith("_info"):
    context = "model_selection"
    current_brand = predicted_tag.replace("_info", "").capitalize()
    available_models = get_available_models(current_brand)
    print(response) # "Which Toyota model are you interested in?"
    continue

# Handle model selection
if context == "model_selection":
    if model:
        car_details = get_car_details(current_brand, model)
        if car_details:
            print(f"Here are the details for {current_brand} {model}:")
            print(f"Price: {car_details['price']} ")
            print(f"Engine: {car_details['engine']} ")
            print(f"Features: {' , '.join(car_details['features'])}")
        else:
            print("Sorry, I couldn't find details for that car.")

    # Ask if the user wants to check another car
    print("Would you like to check another car? (yes/no)")
    context = "ask_another_car"
else:
    available_models = get_available_models(current_brand)
    print(f"Sorry, I don't recognize that {current_brand} model. Please cho
ose from: {' , '.join(available_models)}")

```

```

        continue

    # Handle ask_another_car context
    if context == "ask_another_car":
        if inp.lower() in ["yes", "y"]:
            context = "brand_selection"
            available_brands = get_available_brands()
            print(f"Which car brand are you interested in? Available brands: {', '.join(available_brands)}")
        elif inp.lower() in ["no", "n"]:
            context = None
            print("Okay! Let me know if you need anything else.")
        else:
            print("Please answer with 'yes' or 'no'.")
            continue

    # Default response
    print(response)

# Start chatting
chat()

```

## Key Features

### 1. Dynamic Entity Extraction:

- The chatbot extracts `brand` and `model` entities from user input dynamically.

### 2. Database Integration:

- The chatbot fetches car details from the database based on the extracted entities.

### 3. Interactive Conversation:

- The chatbot resets after providing details and asks if the user wants to check another car.

## Example Conversation

Copy

```

Bot is running! Type 'quit' to exit
You: Hi
Hello!
You: Tell me about Toyota Camry
Here are the details for Toyota Camry:
Price: $25,000
Engine: 2.5L 4-cylinder
Features: Leather Seats, Sunroof, Android Auto

```

```
Would you like to check another car? (yes/no)
You: yes
Which car brand are you interested in? Available brands: Toyota, Honda, Ford, Tesla
You: Tesla Model 3
Here are the details for Tesla Model 3:
Price: $45,000
Engine: Electric
Features: Autopilot, Touchscreen Display, Over-the-Air Updates
Would you like to check another car? (yes/no)
You: no
Okay! Let me know if you need anything else.
You: quit
```

## Next Steps

### 1. Add More Entities:

- Train the model to recognize additional entities like `color`, `price_range`, or `year`.

### 2. Improve Training Data:

- Add more annotated examples to improve the accuracy of entity recognition.

### 3. Enhance UI:

- If you're building a web-based chatbot, display available brands and models as clickable buttons.

Let me know if you need further assistance or enhancements! 😊

in which form the TRAIN DATA is saved?



The `TRAIN_DATA` used for training the entity recognition model is typically saved in a **list of tuples**, where each tuple contains:

1. **Text:** The input text (e.g., "Tell me about Toyota Camry").
2. **Annotations:** A dictionary containing the entity annotations for the text.

Here's the structure of `TRAIN_DATA`:

```
python
```

Copy

```
TRAIN_DATA = [
    (
        "I would like to buy a Toyota Camry",
        {
            "entities": [
                {"start": 0, "end": 6, "entity": "Brand"},  
        ]  
    )  
]
```

```
text_with_entities , # The input text
{
    "entities": [
        (start_index, end_index, "ENTITY_LABEL"), # Entity 1
        (start_index, end_index, "ENTITY_LABEL"), # Entity 2
        ...
    ]
},
...
]
```

---

## Example of `TRAIN_DATA`

Here's an example of what `TRAIN_DATA` might look like:

```
python Copy
TRAIN_DATA = [
(
    "Tell me about Toyota Camry",
    {
        "entities": [
            (16, 22, "BRAND"), # Toyota is a BRAND
            (23, 28, "MODEL") # Camry is a MODEL
        ]
    }
),
(
    "What are the details of Ford Mustang?",
    {
        "entities": [
            (24, 28, "BRAND"), # Ford is a BRAND
            (29, 36, "MODEL") # Mustang is a MODEL
        ]
    }
),
(
    "I want to know about Tesla Model 3",
    {
        "entities": [
            (20, 25, "BRAND"), # Tesla is a BRAND
            (26, 33, "MODEL") # Model 3 is a MODEL
        ]
    }
)
]
```

## Explanation of the Structure

### 1. Text:

- The first element of the tuple is the input text (e.g., "Tell me about Toyota Camry").

### 2. Annotations:

- The second element is a dictionary with a key `"entities"`.
- The value of `"entities"` is a list of tuples, where each tuple represents an entity in the text.

### 3. Entity Tuple:

- Each entity tuple has three elements:
  - **Start Index:** The starting character index of the entity in the text.
  - **End Index:** The ending character index of the entity in the text.
  - **Entity Label:** The label of the entity (e.g., `BRAND`, `MODEL`).

## How to Save `TRAIN_DATA`

You can save `TRAIN_DATA` in a **JSON file** or a **Python script** for later use. Here's how:

### Option 1: Save as a JSON File

```
python                                         Copy
import json

# Save TRAIN_DATA to a JSON file
with open("train_data.json", "w") as file:
    json.dump(TRAIN_DATA, file, indent=4)

print("TRAIN_DATA saved to train_data.json")
```

The `train_data.json` file will look like this:

```
json                                         Copy
[
  [
    [
      "Tell me about Toyota Camry",
      {
        "entities": [
          [16, 22, "BRAND"],
          [23, 28, "MODEL"]
        ]
      }
    ]
]
```

```
        ],
        [
            "What are the details of Ford Mustang?",
            {
                "entities": [
                    [24, 28, "BRAND"],
                    [29, 36, "MODEL"]
                ]
            }
        ],
        [
            "I want to know about Tesla Model 3",
            {
                "entities": [
                    [20, 25, "BRAND"],
                    [26, 33, "MODEL"]
                ]
            }
        ]
    ]
]
```

## Option 2: Save as a Python Script

You can save `TRAIN_DATA` as a Python script for easy reuse:

```
python
# Save TRAIN_DATA to a Python script
with open("train_data.py", "w") as file:
    file.write("TRAIN_DATA = " + repr(TRAIN_DATA))

print("TRAIN_DATA saved to train_data.py")
```

The `train_data.py` file will look like this:

```
python
TRAIN_DATA = [
(
    "Tell me about Toyota Camry",
    {
        "entities": [
            (16, 22, "BRAND"),
            (23, 28, "MODEL")
        ]
    }
),
(
    "What are the details of Ford Mustang?",
```

```
{
    "entities": [
        (24, 28, "BRAND"),
        (29, 36, "MODEL")
    ]
},
(
    "I want to know about Tesla Model 3",
    {
        "entities": [
            (20, 25, "BRAND"),
            (26, 33, "MODEL")
        ]
    }
)
]
```

## How to Load `TRAIN_DATA`

### Option 1: Load from JSON File

```
python                                         Copy

import json

# Load TRAIN_DATA from a JSON file
with open("train_data.json", "r") as file:
    TRAIN_DATA = json.load(file)

print("TRAIN_DATA loaded from train_data.json")
```

### Option 2: Load from Python Script

```
python                                         Copy

# Load TRAIN_DATA from a Python script
from train_data import TRAIN_DATA

print("TRAIN_DATA loaded from train_data.py")
```

## Why Save `TRAIN_DATA`?

### 1. Reusability:

- You can reuse the training data without regenerating it every time.

### 2. Persistence:

## ↳ Persistence:

- The training data is saved permanently and can be shared or version-controlled.

## 3. Efficiency:

- Loading pre-generated training data is faster than generating it from scratch.

## Example Use Case

After saving `TRAIN_DATA`, you can load it and use it to train the entity recognition model:

```
python Copy

import spacy
from spacy.tokens import DocBin
from spacy.training import Example
import random

# Load TRAIN_DATA
with open("train_data.json", "r") as file:
    TRAIN_DATA = json.load(file)

# Load a pre-trained spaCy model
nlp = spacy.load("en_core_web_sm")

# Add entity labels to the pipeline
if "ner" not in nlp.pipe_names:
    ner = nlp.add_pipe("ner")
else:
    ner = nlp.get_pipe("ner")

for _, annotations in TRAIN_DATA:
    for ent in annotations.get("entities"):
        ner.add_label(ent[2])

# Disable other pipeline components during training
other_pipes = [pipe for pipe in nlp.pipe_names if pipe != "ner"]
with nlp.disable_pipes(*other_pipes):
    optimizer = nlp.begin_training()
    for itn in range(10): # Train for 10 iterations
        random.shuffle(TRAIN_DATA)
        losses = {}
        for text, annotations in TRAIN_DATA:
            doc = nlp.make_doc(text)
            example = Example.from_dict(doc, annotations)
            nlp.update([example], drop=0.5, losses=losses)
            print(f"Iteration {itn}, Losses: {losses}")

# Save the trained model
nlp.to_disk("custom_entity_model")
```

## Summary

- **TRAIN\_DATA** is saved as a **list of tuples** containing text and entity annotations.
- You can save it as a **JSON file** or a **Python script** for reuse.
- Loading **TRAIN\_DATA** allows you to train the entity recognition model efficiently.

Let me know if you need further clarification or assistance! 😊

main program giving error `AttributeError: 'NoneType' object has no attribute 'predict'`

The error `AttributeError: 'NoneType' object has no attribute 'predict'` typically occurs when the **model object** is `None` or not properly initialized. This means that the `model` variable is not assigned a valid machine learning model, and when you try to call `model.predict()`, Python raises an error because `None` does not have a `predict` method.

Let's debug and fix this issue step by step.

## Common Causes of the Error

### 1. Model Not Trained:

- The model was not trained properly, so `model` remains `None`.

### 2. Incorrect Model Initialization:

- The model pipeline was not created correctly.

### 3. Data Issues:

- The training data (`X_train` and `y_train`) might be empty or improperly formatted.

### 4. Variable Scope:

- The `model` variable might not be accessible in the scope where `predict` is called.

## Debugging Steps

### Step 1: Check Model Initialization

Ensure that the model is properly initialized and trained. For example:

```
python Copy
# Create a pipeline with TF-IDF and SVM
model = make_pipeline(TfidfVectorizer(), SVC(kernel='linear', probability=True))

# Train the model
model.fit(X_train, y_train)
```

## Step 2: Verify Training Data

Check if `X_train` and `y_train` are properly populated:

```
python Copy
print("X_train:", X_train)
print("y_train:", y_train)
```

If `X_train` or `y_train` is empty, the model will not be trained.

## Step 3: Check Model Object

Before calling `model.predict()`, verify that `model` is not `None`:

```
python Copy
if model is None:
    print("Error: Model is not initialized.")
else:
    predicted_tag = model.predict([processed_input])[0]
```

---

## Updated Code with Debugging

Here's the updated chatbot code with added checks to prevent the `NoneType` error:

```
python Copy
import json
import random
import nltk
import sqlite3
import spacy
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline
```

```

# Download NLTK data
nltk.download('punkt')
nltk.download('wordnet')

# Initialize lemmatizer
lemmatizer = WordNetLemmatizer()

# Load intents file
with open('intents.json') as file:
    intents = json.load(file)

# Load the custom entity recognition model
nlp = spacy.load("custom_entity_model")

# Preprocess data
words = []
classes = []
documents = []
ignore_chars = ['?', '!', '.', ',']

for intent in intents['intents']:
    for pattern in intent['patterns']:
        # Tokenize words
        word_list = nltk.word_tokenize(pattern)
        words.extend(word_list)
        documents.append(( ' '.join(word_list), intent['tag']))
        if intent['tag'] not in classes:
            classes.append(intent['tag'])

# Lemmatize and clean words
words = [lemmatizer.lemmatize(word.lower()) for word in words if word not in ignore_chars]
words = sorted(list(set(words)))
classes = sorted(list(set(classes)))

# Prepare training data
X_train = [] # Patterns
y_train = [] # Intents (tags)

for doc in documents:
    X_train.append(doc[0]) # Pattern
    y_train.append(doc[1]) # Tag

# Verify training data
if not X_train or not y_train:
    raise ValueError("Training data is empty. Check your intents.json file.")

# Create a pipeline with TF-IDF and SVM
model = make_pipeline(TfidfVectorizer(), SVC(kernel='linear', probability=True))

```

```
# Train the model
model.fit(X_train, y_train)

# Verify model initialization
if model is None:
    raise ValueError("Model is not initialized. Check the pipeline creation.")

# Function to fetch car details from the database
def get_car_details(brand, model):
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    cursor.execute('''
        SELECT price, engine, features FROM cars
        WHERE brand = ? AND model = ?
    ''', (brand, model))

    result = cursor.fetchone()
    conn.close()

    if result:
        return {
            "price": result[0],
            "engine": result[1],
            "features": result[2].split(", ")
        }
    else:
        return None

# Function to extract entities from user input
def extract_entities(text):
    doc = nlp(text)
    entities = {}
    for ent in doc.ents:
        entities[ent.label_] = ent.text
    return entities

# Chat function
def chat():
    print("Bot is running! Type 'quit' to exit")
    context = None # To keep track of the conversation context
    current_brand = None # To store the current car brand

    while True:
        inp = input("You: ")
        if inp.lower() == 'quit':
            break

        # Extract entities
        entities = extract_entities(inp)
```

```
brand = entities.get("BRAND")
model_entity = entities.get("MODEL")

# Process input
inp_words = nltk.word_tokenize(inp)
inp_words = [lemmatizer.lemmatize(word.lower()) for word in inp_words]
processed_input = ' '.join(inp_words)

# Predict intent
if model is None:
    print("Error: Model is not initialized.")
    continue

predicted_tag = model.predict([processed_input])[0]

# Get response based on predicted intent
for intent in intents['intents']:
    if intent['tag'] == predicted_tag:
        responses = intent['responses']
        response = random.choice(responses)
        break

# Handle car_info intent
if predicted_tag == "car_info":
    if brand:
        context = "model_selection"
        current_brand = brand
        available_models = get_available_models(current_brand)
        print(f"Which {current_brand} model are you interested in? Available mo
dels: {', '.join(available_models)}")
    else:
        context = "brand_selection"
        available_brands = get_available_brands()
        print(response) # "Which car brand are you interested in? (Toyota, Hon
da, Ford, Tesla)"
        continue

# Handle brand-specific intents (e.g., toyota_info, honda_info)
if predicted_tag.endswith("_info"):
    context = "model_selection"
    current_brand = predicted_tag.replace("_info", "").capitalize()
    available_models = get_available_models(current_brand)
    print(response) # "Which Toyota model are you interested in?"
    continue

# Handle model selection
if context == "model_selection":
    if model_entity:
        car_details = get_car_details(current_brand, model_entity)
        if car_details:
```

```

        print(f"Here are the details for {current_brand} {model_entity}:")
        print(f"Price: {car_details['price']} ")
        print(f"Engine: {car_details['engine']} ")
        print(f"Features: {', '.join(car_details['features'])} ")
    else:
        print("Sorry, I couldn't find details for that car.")

    # Ask if the user wants to check another car
    print("Would you like to check another car? (yes/no)")
    context = "ask_another_car"
else:
    available_models = get_available_models(current_brand)
    print(f"Sorry, I don't recognize that {current_brand} model. Please choose from: {', '.join(available_models)}")
    continue

# Handle ask_another_car context
if context == "ask_another_car":
    if inp.lower() in ["yes", "y"]:
        context = "brand_selection"
        available_brands = get_available_brands()
        print(f"Which car brand are you interested in? Available brands: {', '.join(available_brands)}")
    elif inp.lower() in ["no", "n"]:
        context = None
        print("Okay! Let me know if you need anything else.")
    else:
        print("Please answer with 'yes' or 'no'.")
    continue

# Default response
print(response)

# Start chatting
chat()

```

---

## Key Fixes

### 1. Check for Empty Training Data:

- Added a check to ensure `X_train` and `y_train` are not empty.

### 2. Verify Model Initialization:

- Added a check to ensure `model` is not `None` before calling `model.predict()`.

### 3. Improved Error Handling:

- Added error messages to help debug issues.

## Example Output

If everything is working correctly, the chatbot will run without errors:

```
Copy  
Bot is running! Type 'quit' to exit  
You: Hi  
Hello!  
You: Tell me about Toyota Camry  
Here are the details for Toyota Camry:  
Price: $25,000  
Engine: 2.5L 4-cylinder  
Features: Leather Seats, Sunroof, Android Auto  
Would you like to check another car? (yes/no)  
You: quit
```

## Next Steps

### 1. Check Your Data:

- Ensure `intents.json` and the database are correctly populated.

### 2. Test the Model:

- Verify that the model is trained and can make predictions.

### 3. Add More Features:

- Expand the chatbot to handle more intents and entities.

Let me know if you need further assistance! 😊

it shows me error that get available brand is not there



The error `get_available_brands is not there` occurs because the function `get_available_brands` is either **not defined** or **not accessible** in the scope where it is being called. Let's fix this by ensuring that the function is properly defined and integrated into the chatbot code.

### Step 1: Define `get_available_brands`

The `get_available_brands` function should fetch the list of available car brands from the database. Here's how you can define it:

database. Here's how you can define it:

```
python Copy

def get_available_brands():
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    # Fetch all distinct car brands from the database
    cursor.execute('SELECT DISTINCT brand FROM cars')
    brands = [row[0] for row in cursor.fetchall()]

    conn.close()
    return brands
```

---

## Step 2: Define `get_available_models`

Similarly, the `get_available_models` function should fetch the list of available models for a given brand. Here's how you can define it:

```
python Copy

def get_available_models(brand):
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    # Fetch all models for the given brand
    cursor.execute('SELECT model FROM cars WHERE brand = ?', (brand,))
    models = [row[0] for row in cursor.fetchall()]

    conn.close()
    return models
```

---

## Step 3: Integrate Functions into the Chatbot

Ensure that both `get_available_brands` and `get_available_models` are defined **before** they are used in the `chat` function. Here's the updated chatbot code with these functions integrated:

```
python Copy

import json
import random
import nltk
import sqlite3
import spacy
from nltk.stem import WordNetLemmatizer
```

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline

# Download NLTK data
nltk.download('punkt')
nltk.download('wordnet')

# Initialize lemmatizer
lemmatizer = WordNetLemmatizer()

# Load intents file
with open('intents.json') as file:
    intents = json.load(file)

# Load the custom entity recognition model
nlp = spacy.load("custom_entity_model")

# Preprocess data
words = []
classes = []
documents = []
ignore_chars = ['?', '!', '.', ',']

for intent in intents['intents']:
    for pattern in intent['patterns']:
        # Tokenize words
        word_list = nltk.word_tokenize(pattern)
        words.extend(word_list)
        documents.append(( ' '.join(word_list), intent['tag']))
        if intent['tag'] not in classes:
            classes.append(intent['tag'])

# Lemmatize and clean words
words = [lemmatizer.lemmatize(word.lower()) for word in words if word not in ignore_chars]
words = sorted(list(set(words)))
classes = sorted(list(set(classes)))

# Prepare training data
X_train = [] # Patterns
y_train = [] # Intents (tags)

for doc in documents:
    X_train.append(doc[0]) # Pattern
    y_train.append(doc[1]) # Tag

# Verify training data
if not X_train or not y_train:
    raise ValueError("Training data is empty. Check your intents.json file.")
```

```
# Create a pipeline with TF-IDF and SVM
model = make_pipeline(TfidfVectorizer(), SVC(kernel='linear', probability=True))

# Train the model
model.fit(X_train, y_train)

# Verify model initialization
if model is None:
    raise ValueError("Model is not initialized. Check the pipeline creation.")

# Function to fetch car details from the database
def get_car_details(brand, model):
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    cursor.execute('''
        SELECT price, engine, features FROM cars
        WHERE brand = ? AND model = ?
    ''', (brand, model))

    result = cursor.fetchone()
    conn.close()

    if result:
        return {
            "price": result[0],
            "engine": result[1],
            "features": result[2].split(", ")
        }
    else:
        return None

# Function to fetch available brands from the database
def get_available_brands():
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    # Fetch all distinct car brands from the database
    cursor.execute('SELECT DISTINCT brand FROM cars')
    brands = [row[0] for row in cursor.fetchall()]

    conn.close()
    return brands

# Function to fetch available models for a brand
def get_available_models(brand):
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()
```

```

# Fetch all models for the given brand
cursor.execute('SELECT model FROM cars WHERE brand = ?', (brand,))
models = [row[0] for row in cursor.fetchall()]

conn.close()
return models

# Function to extract entities from user input
def extract_entities(text):
    doc = nlp(text)
    entities = {}
    for ent in doc.ents:
        entities[ent.label_] = ent.text
    return entities

# Chat function
def chat():
    print("Bot is running! Type 'quit' to exit")
    context = None # To keep track of the conversation context
    current_brand = None # To store the current car brand

    while True:
        inp = input("You: ")
        if inp.lower() == 'quit':
            break

        # Extract entities
        entities = extract_entities(inp)
        brand = entities.get("BRAND")
        model_entity = entities.get("MODEL")

        # Process input
        inp_words = nltk.word_tokenize(inp)
        inp_words = [lemmatizer.lemmatize(word.lower()) for word in inp_words]
        processed_input = ' '.join(inp_words)

        # Predict intent
        if model is None:
            print("Error: Model is not initialized.")
            continue

        predicted_tag = model.predict([processed_input])[0]

        # Get response based on predicted intent
        for intent in intents['intents']:
            if intent['tag'] == predicted_tag:
                responses = intent['responses']
                response = random.choice(responses)
                break

```

```
# Handle car_info intent
if predicted_tag == "car_info":
    if brand:
        context = "model_selection"
        current_brand = brand
        available_models = get_available_models(current_brand)
        print(f"Which {current_brand} model are you interested in? Available models: {', '.join(available_models)}")
    else:
        context = "brand_selection"
        available_brands = get_available_brands()
        print(response) # "Which car brand are you interested in? (Toyota, Honda, Ford, Tesla)"
    continue

# Handle brand-specific intents (e.g., toyota_info, honda_info)
if predicted_tag.endswith("_info"):
    context = "model_selection"
    current_brand = predicted_tag.replace("_info", "").capitalize()
    available_models = get_available_models(current_brand)
    print(response) # "Which Toyota model are you interested in?"
    continue

# Handle model selection
if context == "model_selection":
    if model_entity:
        car_details = get_car_details(current_brand, model_entity)
        if car_details:
            print(f"Here are the details for {current_brand} {model_entity}:")
            print(f"Price: {car_details['price']}") 
            print(f"Engine: {car_details['engine']}") 
            print(f"Features: {', '.join(car_details['features'])}")
        else:
            print("Sorry, I couldn't find details for that car.")

    # Ask if the user wants to check another car
    print("Would you like to check another car? (yes/no)")
    context = "ask_another_car"
else:
    available_models = get_available_models(current_brand)
    print(f"Sorry, I don't recognize that {current_brand} model. Please choose from: {', '.join(available_models)}")
    continue

# Handle ask_another_car context
if context == "ask_another_car":
    if inp.lower() in ["yes", "y"]:
        context = "brand_selection"
        available_brands = get_available_brands()
        print(f"Which car brand are you interested in? Available brands: {', '.join(available_brands)}")
```

```
join(available_brands), )
        elif inp.lower() in ["no", "n"]:
            context = None
            print("Okay! Let me know if you need anything else.")
        else:
            print("Please answer with 'yes' or 'no'.")
            continue

        # Default response
        print(response)

# Start chatting
chat()
```

## Key Fixes

1. **Defined** `get_available_brands` **and** `get_available_models`:
  - Both functions are now properly defined and integrated into the chatbot.
2. **Database Integration:**
  - The functions fetch data from the SQLite database (`car_database.db`).
3. **Error Handling:**
  - Added checks to ensure the database connection and queries work correctly.

## Example Output

If everything is set up correctly, the chatbot will work as expected:

Copy

```
Bot is running! Type 'quit' to exit
You: Hi
Hello!
You: Tell me about cars
Which car brand are you interested in? (Toyota, Honda, Ford, Tesla)
You: Toyota
Which Toyota model are you interested in? Available models: Corolla, Camry, Rav4
You: Camry
Here are the details for Toyota Camry:
Price: $25,000
Engine: 2.5L 4-cylinder
Features: Leather Seats, Sunroof, Android Auto
Would you like to check another car? (yes/no)
You: quit
```

## Next Steps

### 1. Check Your Database:

- Ensure the `car_database.db` file exists and contains the `cars` table with `brand` and `model` columns.

### 2. Test the Functions:

- Verify that `get_available_brands` and `get_available_models` return the correct data.

### 3. Expand Functionality:

- Add more features like comparing cars, booking test drives, or finding dealerships.

Let me know if you need further assistance! 😊

NameError: name 'get\_available\_brands' is not defined. Did you mean: 'available\_brands'?



The `NameError: name 'get_available_brands' is not defined` occurs because the function `get_available_brands` is either **not defined** or **not accessible** in the scope where it is being called. This typically happens when:

1. The function is defined **after** it is called.
2. The function is defined inside another function or scope and is not accessible globally.
3. There is a typo in the function name.

Let's fix this issue step by step.

### Step 1: Define `get_available_brands` and `get_available_models`

Ensure that both functions are defined **before** they are used in the `chat` function. Here's how you can define them:

```
python
```

```
Copy
```

```
def get_available_brands():
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    # Fetch all distinct car brands from the database
```

```
// Fetch all distinct car brands from the database
cursor.execute('SELECT DISTINCT brand FROM cars')
brands = [row[0] for row in cursor.fetchall()]

conn.close()
return brands

def get_available_models(brand):
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    # Fetch all models for the given brand
    cursor.execute('SELECT model FROM cars WHERE brand = ?', (brand,))
    models = [row[0] for row in cursor.fetchall()]

    conn.close()
    return models
```

## Step 2: Ensure Functions Are Defined Before Use

Place the definitions of `get_available_brands` and `get_available_models` **before** the `chat` function. Here's the updated structure of your code:

```
python                                         Copy

import json
import random
import nltk
import sqlite3
import spacy
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline

# Download NLTK data
nltk.download('punkt')
nltk.download('wordnet')

# Initialize lemmatizer
lemmatizer = WordNetLemmatizer()

# Load intents file
with open('intents.json') as file:
    intents = json.load(file)

# Load the custom entity recognition model
nlp = spacy.load("custom_entity_model")
```

```

# Preprocess data
words = []
classes = []
documents = []
ignore_chars = ['?', '!', '.', ',']

for intent in intents['intents']:
    for pattern in intent['patterns']:
        # Tokenize words
        word_list = nltk.word_tokenize(pattern)
        words.extend(word_list)
        documents.append(' '.join(word_list), intent['tag']))
        if intent['tag'] not in classes:
            classes.append(intent['tag'])

# Lemmatize and clean words
words = [lemmatizer.lemmatize(word.lower()) for word in words if word not in ignore_chars]
words = sorted(list(set(words)))
classes = sorted(list(set(classes)))

# Prepare training data
X_train = [] # Patterns
y_train = [] # Intents (tags)

for doc in documents:
    X_train.append(doc[0]) # Pattern
    y_train.append(doc[1]) # Tag

# Verify training data
if not X_train or not y_train:
    raise ValueError("Training data is empty. Check your intents.json file.")

# Create a pipeline with TF-IDF and SVM
model = make_pipeline(TfidfVectorizer(), SVC(kernel='linear', probability=True))

# Train the model
model.fit(X_train, y_train)

# Verify model initialization
if model is None:
    raise ValueError("Model is not initialized. Check the pipeline creation.")

# Function to fetch car details from the database
def get_car_details(brand, model):
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    cursor.execute('''
        SELECT price, engine, features FROM cars
    ''')

```

```
WHERE brand = ? AND model = ?
'', (brand, model))

result = cursor.fetchone()
conn.close()

if result:
    return {
        "price": result[0],
        "engine": result[1],
        "features": result[2].split(", ")
    }
else:
    return None

# Function to fetch available brands from the database
def get_available_brands():
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    # Fetch all distinct car brands from the database
    cursor.execute('SELECT DISTINCT brand FROM cars')
    brands = [row[0] for row in cursor.fetchall()]

    conn.close()
    return brands

# Function to fetch available models for a brand
def get_available_models(brand):
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    # Fetch all models for the given brand
    cursor.execute('SELECT model FROM cars WHERE brand = ?', (brand,))
    models = [row[0] for row in cursor.fetchall()]

    conn.close()
    return models

# Function to extract entities from user input
def extract_entities(text):
    doc = nlp(text)
    entities = {}
    for ent in doc.ents:
        entities[ent.label_] = ent.text
    return entities

# Chat function
def chat():
    print("Bot is running! Type 'quit' to exit")
```

```
context = None # To keep track of the conversation context
current_brand = None # To store the current car brand

while True:
    inp = input("You: ")
    if inp.lower() == 'quit':
        break

    # Extract entities
    entities = extract_entities(inp)
    brand = entities.get("BRAND")
    model_entity = entities.get("MODEL")

    # Process input
    inp_words = nltk.word_tokenize(inp)
    inp_words = [lemmatizer.lemmatize(word.lower()) for word in inp_words]
    processed_input = ' '.join(inp_words)

    # Predict intent
    if model is None:
        print("Error: Model is not initialized.")
        continue

    predicted_tag = model.predict([processed_input])[0]

    # Get response based on predicted intent
    for intent in intents['intents']:
        if intent['tag'] == predicted_tag:
            responses = intent['responses']
            response = random.choice(responses)
            break

    # Handle car_info intent
    if predicted_tag == "car_info":
        if brand:
            context = "model_selection"
            current_brand = brand
            available_models = get_available_models(current_brand)
            print(f"Which {current_brand} model are you interested in? Available models: {', '.join(available_models)}")
        else:
            context = "brand_selection"
            available_brands = get_available_brands()
            print(response) # "Which car brand are you interested in? (Toyota, Honda, Ford, Tesla)"
            continue

    # Handle brand-specific intents (e.g., toyota_info, honda_info)
    if predicted_tag.endswith("_info"):
        context = "model_selection"
```

```

        current_brand = predicted_tag.replace( _intro ,    ).capitalize()
        available_models = get_available_models(current_brand)
        print(response) # "Which Toyota model are you interested in?"
        continue

        # Handle model selection
        if context == "model_selection":
            if model_entity:
                car_details = get_car_details(current_brand, model_entity)
                if car_details:
                    print(f"Here are the details for {current_brand} {model_entity}:")
                    print(f"Price: {car_details['price']} ")
                    print(f"Engine: {car_details['engine']} ")
                    print(f"Features: {', '.join(car_details['features'])}")
                else:
                    print("Sorry, I couldn't find details for that car.")

            # Ask if the user wants to check another car
            print("Would you like to check another car? (yes/no)")
            context = "ask_another_car"
        else:
            available_models = get_available_models(current_brand)
            print(f"Sorry, I don't recognize that {current_brand} model. Please choose from: {', '.join(available_models)}")
            continue

        # Handle ask_another_car context
        if context == "ask_another_car":
            if inp.lower() in ["yes", "y"]:
                context = "brand_selection"
                available_brands = get_available_brands()
                print(f"Which car brand are you interested in? Available brands: {', '.join(available_brands)}")
            elif inp.lower() in ["no", "n"]:
                context = None
                print("Okay! Let me know if you need anything else.")
            else:
                print("Please answer with 'yes' or 'no'.")
            continue

        # Default response
        print(response)

# Start chatting
chat()

```

---

## Key Fixes

### 1. Function Definitions:

- `get_available_brands` and `get_available_models` are defined **before** they are used in the `chat` function.

## 2. Database Integration:

- Both functions fetch data from the SQLite database (`car_database.db`).

## 3. Error Handling:

- Added checks to ensure the database connection and queries work correctly.

## Example Output

If everything is set up correctly, the chatbot will work as expected:

```
Copy

Bot is running! Type 'quit' to exit
You: Hi
Hello!
You: Tell me about cars
Which car brand are you interested in? (Toyota, Honda, Ford, Tesla)
You: Toyota
Which Toyota model are you interested in? Available models: Corolla, Camry, Rav4
You: Camry
Here are the details for Toyota Camry:
Price: $25,000
Engine: 2.5L 4-cylinder
Features: Leather Seats, Sunroof, Android Auto
Would you like to check another car? (yes/no)
You: quit
```

## Next Steps

### 1. Check Your Database:

- Ensure the `car_database.db` file exists and contains the `cars` table with `brand` and `model` columns.

### 2. Test the Functions:

- Verify that `get_available_brands` and `get_available_models` return the correct data.

### 3. Expand Functionality:

- Add more features like comparing cars, booking test drives, or finding dealerships.

Let me know if you need further assistance! 😊

but if i enter toyota camry in one line will it give answer?



Yes, if you enter "**Toyota Camry**" in one line, the chatbot should recognize both the **brand** (**Toyota**) and the **model** (**Camry**) using the **entity recognition model** and provide the details of the car. This is because the entity recognition model is trained to extract **BRAND** and **MODEL** entities from user input.

Let's walk through how this works and ensure the chatbot handles such inputs correctly.

---

## How It Works

### 1. Entity Recognition:

- The chatbot uses the trained spaCy model to extract **BRAND** and **MODEL** entities from the user input.
- For example, if the input is **"Tell me about Toyota Camry"**, the model will recognize:
  - **Toyota** as **BRAND**
  - **Camry** as **MODEL**

### 2. Fetch Car Details:

- Once the entities are extracted, the chatbot fetches the car details from the database using the **get\_car\_details** function.

### 3. Provide Response:

- The chatbot displays the car details (price, engine, features) to the user.
- 

## Updated Chatbot Code

Here's the updated chatbot code to ensure it handles single-line inputs like **"Toyota Camry"** :

```
python
```

```
Copy
```

```
import json
import random
import nltk
import sqlite3
```

```
import spacy
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline

# Download NLTK data
nltk.download('punkt')
nltk.download('wordnet')

# Initialize lemmatizer
lemmatizer = WordNetLemmatizer()

# Load intents file
with open('intents.json') as file:
    intents = json.load(file)

# Load the custom entity recognition model
nlp = spacy.load("custom_entity_model")

# Preprocess data
words = []
classes = []
documents = []
ignore_chars = ['?', '!', '.', ',']

for intent in intents['intents']:
    for pattern in intent['patterns']:
        # Tokenize words
        word_list = nltk.word_tokenize(pattern)
        words.extend(word_list)
        documents.append(' '.join(word_list), intent['tag']))
        if intent['tag'] not in classes:
            classes.append(intent['tag'])

# Lemmatize and clean words
words = [lemmatizer.lemmatize(word.lower()) for word in words if word not in ignore_chars]
words = sorted(list(set(words)))
classes = sorted(list(set(classes)))

# Prepare training data
X_train = [] # Patterns
y_train = [] # Intents (tags)

for doc in documents:
    X_train.append(doc[0]) # Pattern
    y_train.append(doc[1]) # Tag

# Verify training data
```

```
if not X_train or not y_train:
    raise ValueError("Training data is empty. Check your intents.json file.")

# Create a pipeline with TF-IDF and SVM
model = make_pipeline(TfidfVectorizer(), SVC(kernel='linear', probability=True))

# Train the model
model.fit(X_train, y_train)

# Verify model initialization
if model is None:
    raise ValueError("Model is not initialized. Check the pipeline creation.")

# Function to fetch car details from the database
def get_car_details(brand, model):
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    cursor.execute('''
        SELECT price, engine, features FROM cars
        WHERE brand = ? AND model = ?
    ''', (brand, model))

    result = cursor.fetchone()
    conn.close()

    if result:
        return {
            "price": result[0],
            "engine": result[1],
            "features": result[2].split(", ")
        }
    else:
        return None

# Function to fetch available brands from the database
def get_available_brands():
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    # Fetch all distinct car brands from the database
    cursor.execute('SELECT DISTINCT brand FROM cars')
    brands = [row[0] for row in cursor.fetchall()]

    conn.close()
    return brands

# Function to fetch available models for a brand
def get_available_models(brand):
    conn = sqlite3.connect('car_database.db')
    cursor = conn.cursor()

    cursor.execute('SELECT model FROM cars WHERE brand = ?',
    
```

```
cursor = conn.cursor()

# Fetch all models for the given brand
cursor.execute('SELECT model FROM cars WHERE brand = ? ', (brand,))
models = [row[0] for row in cursor.fetchall()]

conn.close()
return models

# Function to extract entities from user input
def extract_entities(text):
    doc = nlp(text)
    entities = {}
    for ent in doc.ents:
        entities[ent.label_] = ent.text
    return entities

# Chat function
def chat():
    print("Bot is running! Type 'quit' to exit")
    context = None # To keep track of the conversation context
    current_brand = None # To store the current car brand

    while True:
        inp = input("You: ")
        if inp.lower() == 'quit':
            break

        # Extract entities
        entities = extract_entities(inp)
        brand = entities.get("BRAND")
        model_entity = entities.get("MODEL")

        # Process input
        inp_words = nltk.word_tokenize(inp)
        inp_words = [lemmatizer.lemmatize(word.lower()) for word in inp_words]
        processed_input = ' '.join(inp_words)

        # Predict intent
        if model is None:
            print("Error: Model is not initialized.")
            continue

        predicted_tag = model.predict([processed_input])[0]

        # Get response based on predicted intent
        for intent in intents['intents']:
            if intent['tag'] == predicted_tag:
                responses = intent['responses']
                response = random.choice(responses)
                break
```

```
break

# Handle car_info intent
if predicted_tag == "car_info":
    if brand and model_entity:
        # If both brand and model are provided, fetch details directly
        car_details = get_car_details(brand, model_entity)
        if car_details:
            print(f"Here are the details for {brand} {model_entity}:")
            print(f"Price: {car_details['price']} ")
            print(f"Engine: {car_details['engine']} ")
            print(f"Features: {', '.join(car_details['features'])}")
        else:
            print("Sorry, I couldn't find details for that car.")

    # Ask if the user wants to check another car
    print("Would you like to check another car? (yes/no)")
    context = "ask_another_car"
elif brand:
    # If only brand is provided, ask for the model
    context = "model_selection"
    current_brand = brand
    available_models = get_available_models(current_brand)
    print(f"Which {current_brand} model are you interested in? Available models: {', '.join(available_models)}")
else:
    # If no brand is provided, ask for the brand
    context = "brand_selection"
    available_brands = get_available_brands()
    print(response) # "Which car brand are you interested in? (Toyota, Honda, Ford, Tesla)"
    continue

# Handle brand-specific intents (e.g., toyota_info, honda_info)
if predicted_tag.endswith("_info"):
    context = "model_selection"
    current_brand = predicted_tag.replace("_info", "").capitalize()
    available_models = get_available_models(current_brand)
    print(response) # "Which Toyota model are you interested in?"
    continue

# Handle model selection
if context == "model_selection":
    if model_entity:
        car_details = get_car_details(current_brand, model_entity)
        if car_details:
            print(f"Here are the details for {current_brand} {model_entity}:")
            print(f"Price: {car_details['price']} ")
            print(f"Engine: {car_details['engine']} ")
            print(f"Features: {', '.join(car_details['features'])}")
        else:
```

```

        print("Sorry, I couldn't find details for that car.")

        # Ask if the user wants to check another car
        print("Would you like to check another car? (yes/no)")
        context = "ask_another_car"
    else:
        available_models = get_available_models(current_brand)
        print(f"Sorry, I don't recognize that {current_brand} model. Please choose from: {', '.join(available_models)}")
        continue

    # Handle ask_another_car context
    if context == "ask_another_car":
        if inp.lower() in ["yes", "y"]:
            context = "brand_selection"
            available_brands = get_available_brands()
            print(f"Which car brand are you interested in? Available brands: {', '.join(available_brands)}")
        elif inp.lower() in ["no", "n"]:
            context = None
            print("Okay! Let me know if you need anything else.")
        else:
            print("Please answer with 'yes' or 'no'.")
        continue

    # Default response
    print(response)

# Start chatting
chat()

```

## Key Changes

### 1. Direct Handling of Brand and Model:

- If both `brand` and `model` are provided in the input (e.g., `"Toyota Camry"`), the chatbot fetches and displays the car details directly.

### 2. Context Handling:

- If only the `brand` is provided, the chatbot asks for the `model`.
- If neither is provided, the chatbot asks for the `brand`.

### 3. Entity Recognition:

- The chatbot uses the trained spaCy model to extract `BRAND` and `MODEL` entities from the input.

## Example Conversation

Copy

```
Bot is running! Type 'quit' to exit
You: Hi
Hello!
You: Tell me about Toyota Camry
Here are the details for Toyota Camry:
Price: $25,000
Engine: 2.5L 4-cylinder
Features: Leather Seats, Sunroof, Android Auto
Would you like to check another car? (yes/no)
You: no
Okay! Let me know if you need anything else.
You: quit
```

## Next Steps

### 1. Test the Chatbot:

- Try inputs like "Toyota Camry" , "Tell me about Ford Mustang" , or "What are the details of Tesla Model 3?" .

### 2. Improve Entity Recognition:

- Add more training data to improve the accuracy of entity extraction.

### 3. Expand Functionality:

- Add more features like comparing cars, booking test drives, or finding dealerships.

Let me know if you need further assistance! 😊

this entity recognition is not working in the chatbot



If the **entity recognition** is not working in your chatbot, it could be due to several reasons, such as:

### 1. Incorrect Training Data:

- The `TRAIN_DATA` used to train the entity recognition model might not be sufficient or properly annotated.

### 2. Model Not Trained Properly:

- The spaCy model might not have been trained correctly or saved/loaded properly.

### 3. Entity Extraction Logic:

- The logic for extracting entities from user input might be flawed.

### 4. Database Issues:

- The database might not contain the expected data, or the queries might not be working as intended.

Let's debug and fix the issue step by step.

---

## Step 1: Verify Entity Recognition Model

### Check Training Data

Ensure that the `TRAIN_DATA` used to train the entity recognition model is correct. For example:

```
python                                         Copy

TRAIN_DATA = [
    ("Tell me about Toyota Camry", {"entities": [(16, 22, "BRAND"), (23, 28, "MODEL")]}),
    ("What are the details of Ford Mustang?", {"entities": [(24, 28, "BRAND"), (29, 36, "MODEL")]}),
    ("I want to know about Tesla Model 3", {"entities": [(20, 25, "BRAND"), (26, 33, "MODEL")]}),
]
```

### Train and Save the Model

Ensure that the model is trained and saved correctly:

```
python                                         Copy

import spacy
from spacy.tokens import DocBin
from spacy.training import Example
import random

# Load a pre-trained spaCy model
nlp = spacy.load("en_core_web_sm")

# Add entity labels to the pipeline
if "ner" not in nlp.pipe_names:
    ner = nlp.add_pipe("ner")
else:
```

```
ner = nlp.get_pipe("ner")

for _, annotations in TRAIN_DATA:
    for ent in annotations.get("entities"):
        ner.add_label(ent[2])

# Disable other pipeline components during training
other_pipes = [pipe for pipe in nlp.pipe_names if pipe != "ner"]
with nlp.disable_pipes(*other_pipes):
    optimizer = nlp.begin_training()
    for itn in range(10): # Train for 10 iterations
        random.shuffle(TRAIN_DATA)
        losses = {}
        for text, annotations in TRAIN_DATA:
            doc = nlp.make_doc(text)
            example = Example.from_dict(doc, annotations)
            nlp.update([example], drop=0.5, losses=losses)
        print(f"Iteration {itn}, Losses: {losses}")

# Save the trained model
nlp.to_disk("custom_entity_model")
```

## Load the Model

Ensure that the model is loaded correctly in the chatbot:

```
python
nlp = spacy.load("custom_entity_model")
```

## Step 2: Debug Entity Extraction

Add debug statements to check if entities are being extracted correctly:

```
python
def extract_entities(text):
    doc = nlp(text)
    entities = {}
    for ent in doc.ents:
        print(f"Entity: {ent.text}, Label: {ent.label_}") # Debug statement
        entities[ent.label_] = ent.text
    return entities
```

## Step 3: Update Chatbot Logic

Ensure that the chatbot logic correctly handles the extracted entities. Here's the updated

## chat function:

python

Copy

```
def chat():
    print("Bot is running! Type 'quit' to exit")
    context = None # To keep track of the conversation context
    current_brand = None # To store the current car brand

    while True:
        inp = input("You: ")
        if inp.lower() == 'quit':
            break

        # Extract entities
        entities = extract_entities(inp)
        print("Extracted Entities:", entities) # Debug statement
        brand = entities.get("BRAND")
        model_entity = entities.get("MODEL")

        # Process input
        inp_words = nltk.word_tokenize(inp)
        inp_words = [lemmatizer.lemmatize(word.lower()) for word in inp_words]
        processed_input = ' '.join(inp_words)

        # Predict intent
        if model is None:
            print("Error: Model is not initialized.")
            continue

        predicted_tag = model.predict([processed_input])[0]

        # Get response based on predicted intent
        for intent in intents['intents']:
            if intent['tag'] == predicted_tag:
                responses = intent['responses']
                response = random.choice(responses)
                break

        # Handle car_info intent
        if predicted_tag == "car_info":
            if brand and model_entity:
                # If both brand and model are provided, fetch details directly
                car_details = get_car_details(brand, model_entity)
                if car_details:
                    print(f"Here are the details for {brand} {model_entity}:")
                    print(f"Price: {car_details['price']}")  
print(f"Engine: {car_details['engine']}")  
print(f"Features: {', '.join(car_details['features'])}")  
else:  
    print("No details found for the specified car brand and model.")
```

```
        print("Sorry, I couldn't find details for that car.")

        # Ask if the user wants to check another car
        print("Would you like to check another car? (yes/no)")
        context = "ask_another_car"

    elif brand:
        # If only brand is provided, ask for the model
        context = "model_selection"
        current_brand = brand
        available_models = get_available_models(current_brand)
        print(f"Which {current_brand} model are you interested in? Available models: {', '.join(available_models)}")
    else:
        # If no brand is provided, ask for the brand
        context = "brand_selection"
        available_brands = get_available_brands()
        print(response) # "Which car brand are you interested in? (Toyota, Honda, Ford, Tesla)"
        continue

    # Handle brand-specific intents (e.g., toyota_info, honda_info)
    if predicted_tag.endswith("_info"):
        context = "model_selection"
        current_brand = predicted_tag.replace("_info", "").capitalize()
        available_models = get_available_models(current_brand)
        print(response) # "Which Toyota model are you interested in?"
        continue

    # Handle model selection
    if context == "model_selection":
        if model_entity:
            car_details = get_car_details(current_brand, model_entity)
            if car_details:
                print(f"Here are the details for {current_brand} {model_entity}:")
                print(f"Price: {car_details['price']}")
                print(f"Engine: {car_details['engine']}")
                print(f"Features: {', '.join(car_details['features'])}")
            else:
                print("Sorry, I couldn't find details for that car.")

        # Ask if the user wants to check another car
        print("Would you like to check another car? (yes/no)")
        context = "ask_another_car"

    else:
        available_models = get_available_models(current_brand)
        print(f"Sorry, I don't recognize that {current_brand} model. Please choose from: {', '.join(available_models)}")
        continue

    # Handle ask_another_car context
```

```
if context == "ask_another_car":
    if inp.lower() in ["yes", "y"]:
        context = "brand_selection"
        available_brands = get_available_brands()
        print(f"Which car brand are you interested in? Available brands: {', '.join(available_brands)}")
    elif inp.lower() in ["no", "n"]:
        context = None
        print("Okay! Let me know if you need anything else.")
    else:
        print("Please answer with 'yes' or 'no'.")
        continue

# Default response
print(response)
```

## Step 4: Test the Chatbot

Run the chatbot and test it with inputs like:

- "Tell me about Toyota Camry"
- "What are the details of Ford Mustang?"
- "I want to know about Tesla Model 3"

Check the debug statements to ensure entities are being extracted correctly.

## Example Debug Output

If the entity recognition is working, you should see output like this:

```
You: Tell me about Toyota Camry
Extracted Entities: {'BRAND': 'Toyota', 'MODEL': 'Camry'}
Here are the details for Toyota Camry:
Price: $25,000
Engine: 2.5L 4-cylinder
Features: Leather Seats, Sunroof, Android Auto
Would you like to check another car? (yes/no)
```

Copy

## Common Issues and Fixes

### 1. No Entities Extracted:

- Ensure the `TRAIN_DATA` is correct and the model is trained properly.

- Add more training examples if necessary.

## 2. Incorrect Entities Extracted:

- Check the entity annotations in `TRAIN_DATA`.
- Ensure the model is trained with sufficient iterations.

## 3. Database Issues:

- Verify that the database contains the expected data and the queries are correct.

---

Let me know if you need further assistance! 😊



New chat

Message DeepSeek

DeepThink (R1)

Search



AI-generated, for reference only

