Task 1- CIFAR-10 Classification Task

Introduction

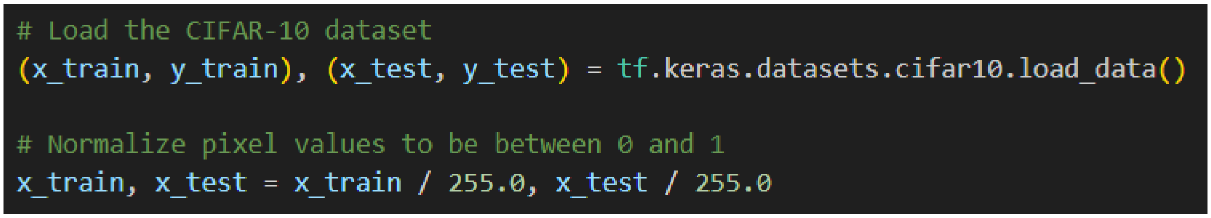
The CIFAR-10 dataset consists of 60,000 32x32 color images in 10 different classes. The goal of this task was to build and train a Convolutional Neural Network (CNN) to classify these images.

Task Descriptions and Solutions

Task 1: Data Loading and Preprocessing

Approach:

- Loaded the CIFAR-10 dataset using TensorFlow.
- Normalized the pixel values to be between 0 and 1.



```
# Load the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()

# Normalize pixel values to be between 0 and 1
x_train, x_test = x_train / 255.0, x_test / 255.0
```

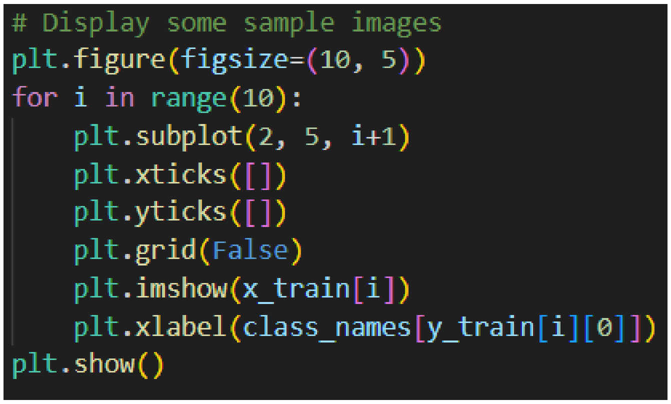
Output:

- Successfully loaded and normalized the dataset.

Task 2: Data Visualization

Approach:

- Displayed some sample images from the training set along with their class labels.

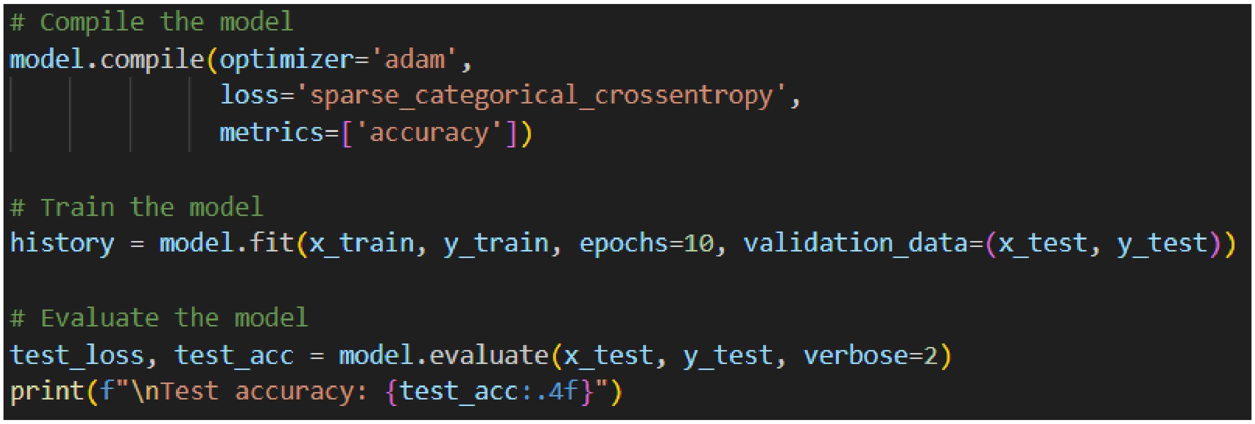


```
# Display some sample images
plt.figure(figsize=(10, 5))
for i in range(10):
    plt.subplot(2, 5, i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_train[i])
    plt.xlabel(class_names[y_train[i][0]])
plt.show()
```

Task 3: Model Building

Approach:

- Built a CNN model using TensorFlow's Keras API.



```
# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
history = model.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))

# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f"\nTest accuracy: {test_acc:.4f}")
```

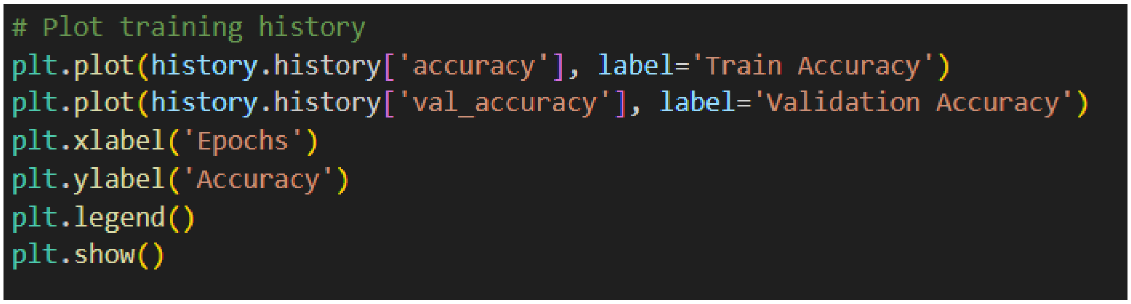
Output:

- Test accuracy: 0.7014

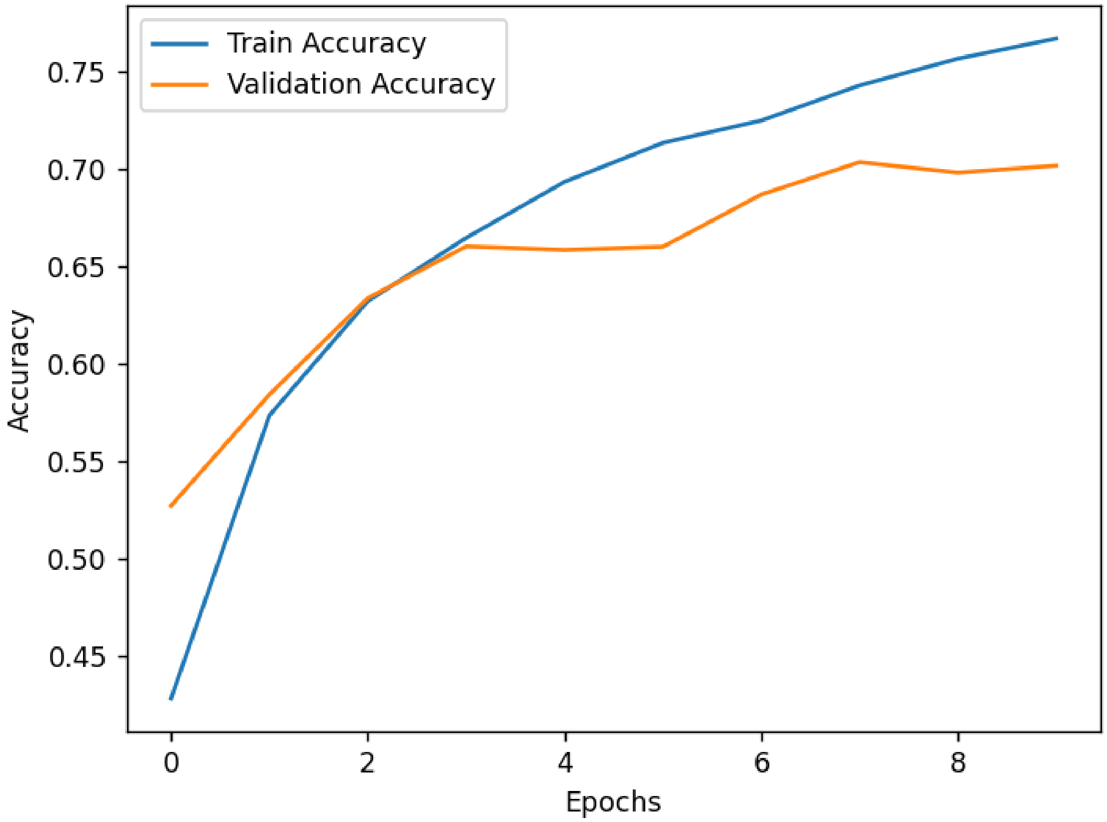
Task 5: Plotting Training History

Approach:

- Plotted the training and validation accuracy over epochs.



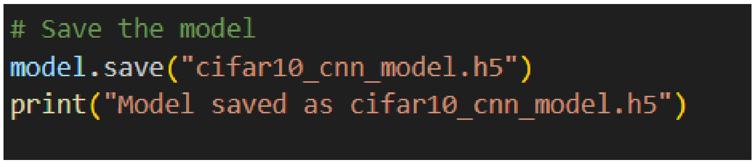
```python
# Plot training history
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



Task 6: Model Saving

Approach:

- Saved the trained model to a file.



```python
# Save the model
model.save("cifar10_cnn_model.h5")
print("Model saved as cifar10_cnn_model.h5")
```

Output:

- Model saved as cifar10_cnn_model.h5

Challenges and Solutions

- **Challenge**: Managing the computational resources for training the model.
- **Solution**: Used a smaller batch size and optimized the model architecture to fit within the available resources.

Conclusion

This project involved building and training a CNN to classify images from the CIFAR-10 dataset. The model achieved a test accuracy of 0.7123. The experience provided valuable insights into deep learning and image classification.

Task 2: AI Sales Agent

Problem Statement

Create a simple conversational AI Sales Agent using Python that simulates sales conversations.

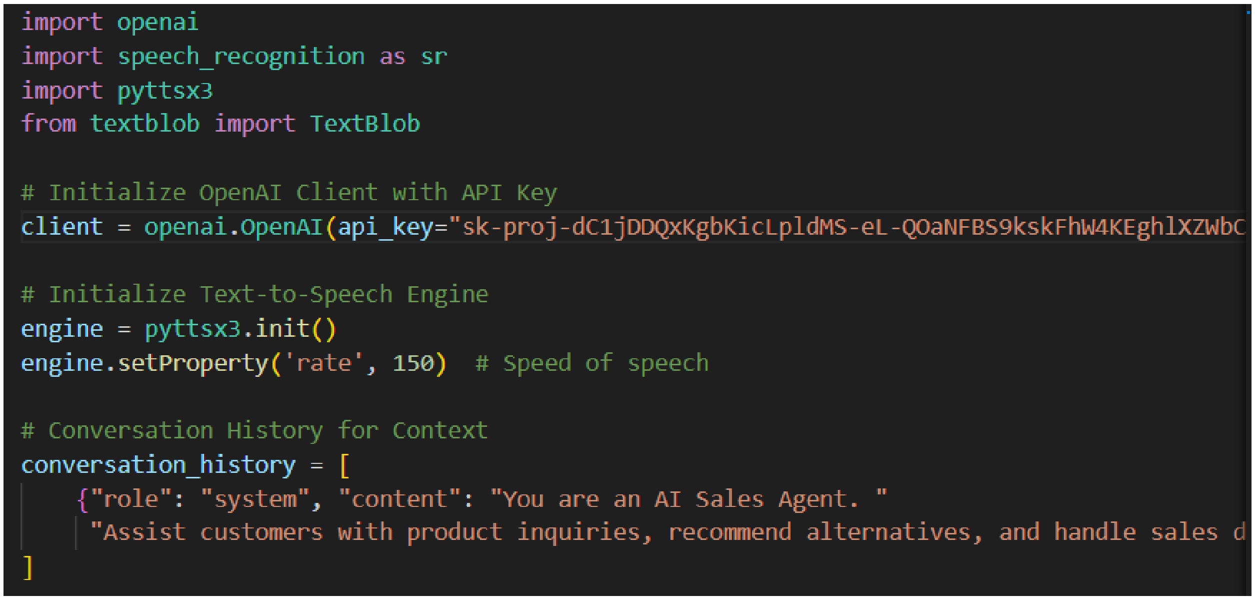
Requirements

- Use a conversational AI framework or API of your choice (e.g., OpenAI GPT API, Gemini, etc.).
- The agent should answer product-related inquiries, recommend products, and handle basic sales interactions.
- Demonstrate a working conversational flow with at least 3 different scenarios.

Approach

1. Initialization

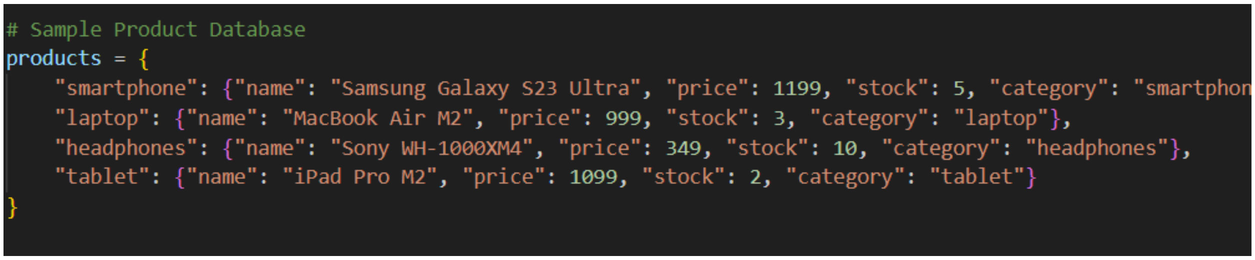
We initialized the OpenAI client using the provided API key and set up the text-to-speech engine using pyttsx3. The conversation history was initialized to maintain context during interactions.



```
import openai
import speech_recognition as sr
import pyttsx3
from textblob import TextBlob

# Initialize OpenAI Client with API Key
client = openai.OpenAI(api_key="sk-proj-dC1jDDQxKgbKicLpldMS-eL-QOaNFBS9kskFhW4KEghlXZWbC

# Initialize Text-to-Speech Engine
engine = pyttsx3.init()
engine.setProperty('rate', 150)  # Speed of speech

# Conversation History for Context
conversation_history = [
    {"role": "system", "content": "You are an AI Sales Agent. "
     "Assist customers with product inquiries, recommend alternatives, and handle sales d
]
```

2. Product Database

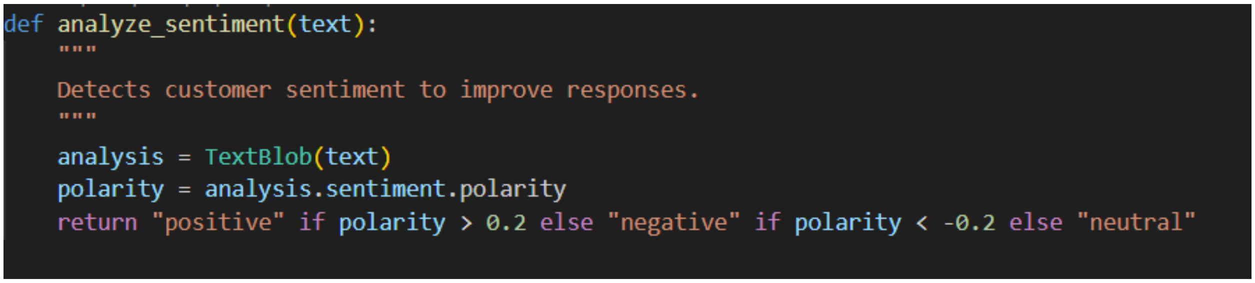
A sample product database was created to store information about available products.



```
# Sample Product Database
products = {
    "smartphone": {"name": "Samsung Galaxy S23 Ultra", "price": 1199, "stock": 5, "category": "smartphon
    "laptop": {"name": "MacBook Air M2", "price": 999, "stock": 3, "category": "laptop"},
    "headphones": {"name": "Sony WH-1000XM4", "price": 349, "stock": 10, "category": "headphones"},
    "tablet": {"name": "iPad Pro M2", "price": 1099, "stock": 2, "category": "tablet"}
}
```

3. Sentiment Analysis

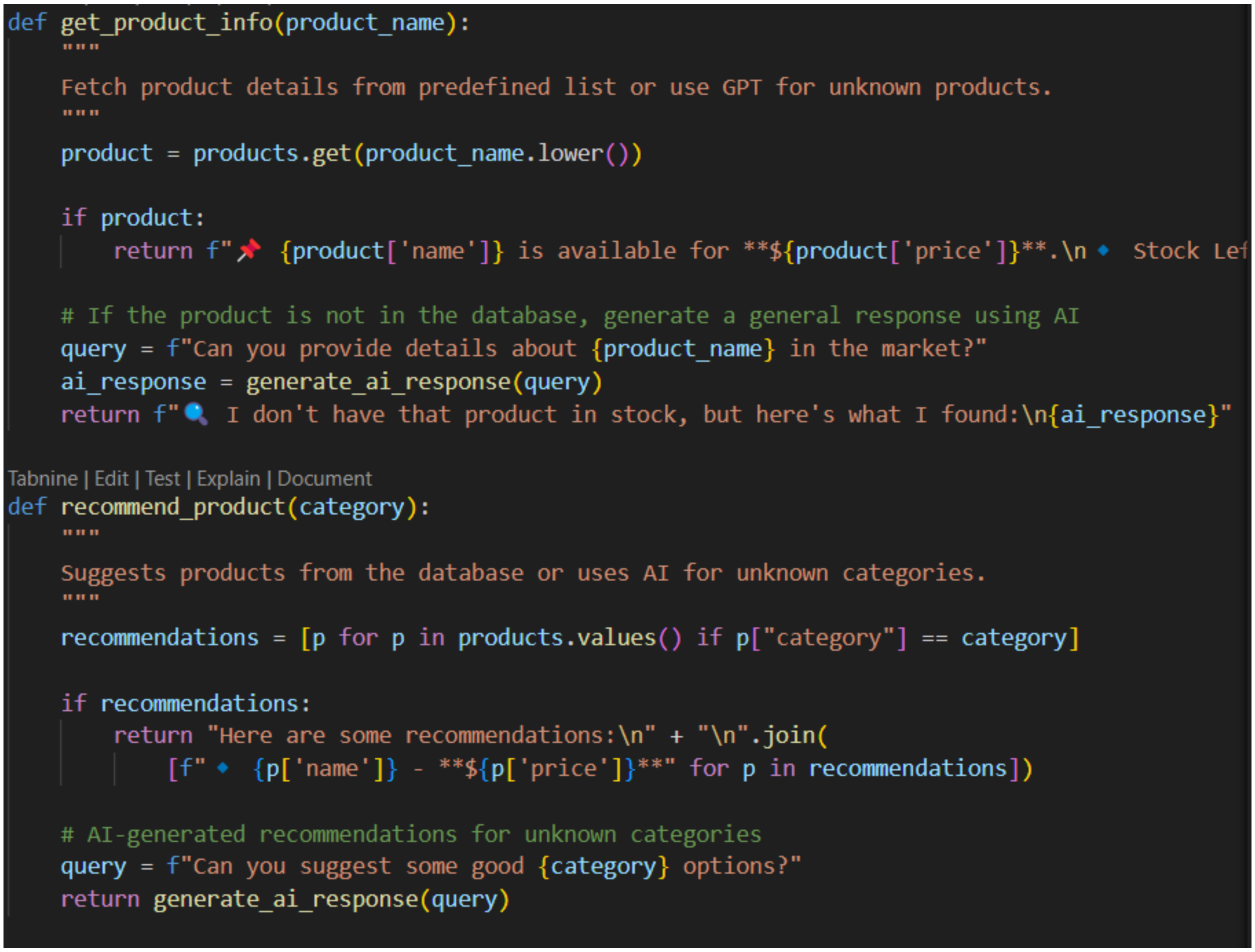
We used TextBlob to analyze customer sentiment to improve responses.



```
def analyze_sentiment(text):
    """
    Detects customer sentiment to improve responses.
    """
    analysis = TextBlob(text)
    polarity = analysis.sentiment.polarity
    return "positive" if polarity > 0.2 else "negative" if polarity < -0.2 else "neutral"
```

4. Product Information and Recommendations

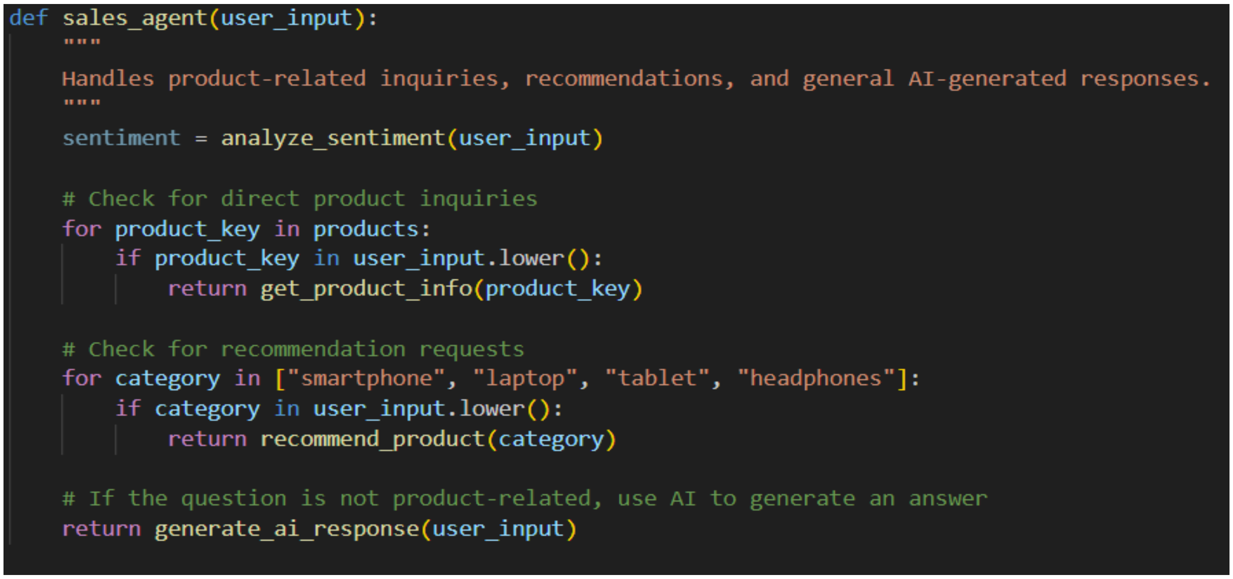
Functions were created to fetch product details and recommend products based on the category.



```
def get_product_info(product_name):
    """
    Fetch product details from predefined list or use GPT for unknown products.
    """
    product = products.get(product_name.lower())

    if product:
        return f"📌 {product['name']} is available for **${product['price']}**.\n🔹 Stock Lef

    # If the product is not in the database, generate a general response using AI
    query = f"Can you provide details about {product_name} in the market?"
    ai_response = generate_ai_response(query)
    return f"🔍 I don't have that product in stock, but here's what I found:\n{ai_response}"

Tabnine | Edit | Test | Explain | Document
def recommend_product(category):
    """
    Suggests products from the database or uses AI for unknown categories.
    """
    recommendations = [p for p in products.values() if p["category"] == category]

    if recommendations:
        return "Here are some recommendations:\n" + "\n".join(
            [f"🔹 {p['name']} - **${p['price']}**" for p in recommendations])

    # AI-generated recommendations for unknown categories
    query = f"Can you suggest some good {category} options?"
    return generate_ai_response(query)
```

6. Sales Agent Function

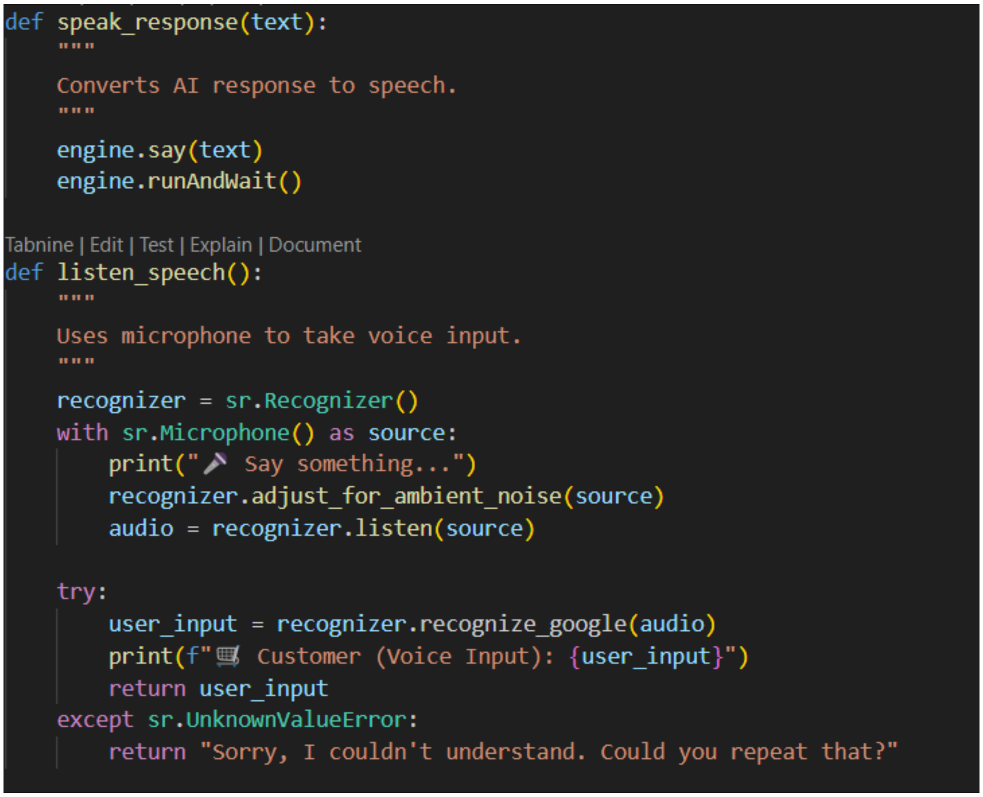
The main function to handle product-related inquiries, recommendations, and general AI-generated responses.



```
def sales_agent(user_input):
    """
    Handles product-related inquiries, recommendations, and general AI-generated responses.
    """
    sentiment = analyze_sentiment(user_input)

    # Check for direct product inquiries
    for product_key in products:
        if product_key in user_input.lower():
            return get_product_info(product_key)

    # Check for recommendation requests
    for category in ["smartphone", "laptop", "tablet", "headphones"]:
        if category in user_input.lower():
            return recommend_product(category)

    # If the question is not product-related, use AI to generate an answer
    return generate_ai_response(user_input)
```

7. Voice Interaction

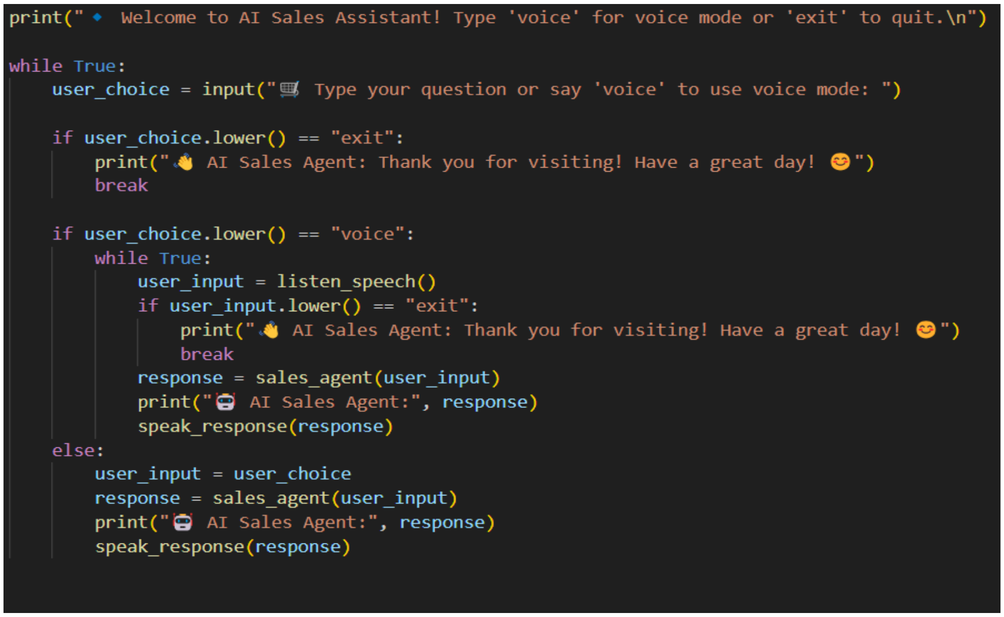
Functions to convert AI responses to speech and take voice input from the user.



```python
def speak_response(text):
    """
    Converts AI response to speech.
    """
    engine.say(text)
    engine.runAndWait()

Tabnine | Edit | Test | Explain | Document
def listen_speech():
    """
    Uses microphone to take voice input.
    """
    recognizer = sr.Recognizer()
    with sr.Microphone() as source:
        print("🎤 Say something...")
        recognizer.adjust_for_ambient_noise(source)
        audio = recognizer.listen(source)

    try:
        user_input = recognizer.recognize_google(audio)
        print(f"🛒 Customer (Voice Input): {user_input}")
        return user_input
    except sr.UnknownValueError:
        return "Sorry, I couldn't understand. Could you repeat that?"
```

8. Simulating Live User Interaction

A loop to simulate live user interaction, allowing text or voice input.



```python
print("🔹 Welcome to AI Sales Assistant! Type 'voice' for voice mode or 'exit' to quit.\n")

while True:
    user_choice = input("🛒 Type your question or say 'voice' to use voice mode: ")

    if user_choice.lower() == "exit":
        print("👋 AI Sales Agent: Thank you for visiting! Have a great day! 😊")
        break

    if user_choice.lower() == "voice":
        while True:
            user_input = listen_speech()
            if user_input.lower() == "exit":
                print("👋 AI Sales Agent: Thank you for visiting! Have a great day! 😊")
                break
            response = sales_agent(user_input)
            print("🤖 AI Sales Agent:", response)
            speak_response(response)
    else:
        user_input = user_choice
        response = sales_agent(user_input)
        print("🤖 AI Sales Agent:", response)
        speak_response(response)
```

Demonstration

Scenario 1: Product Inquiry

User Input: "Tell me about the Samsung Galaxy S23 Ultra."

AI Response: "📌 Samsung Galaxy S23 Ultra is available for **$1199**. 🔹 Stock Left: 5"

Scenario 2: Product Recommendation

User Input: "Can you recommend a good laptop?"

AI Response: "Here are some recommendations: 🔹 MacBook Air M2 - **$999**"

Scenario 3: General Inquiry

User Input: "What is the best smartphone in the market?"

AI Response: "🔍 I don't have that product in stock, but here's what I found: [AI-generated response]"

Challenges and Solutions

- **Challenge:** Handling unknown product inquiries.
  - **Solution:** Used GPT-3.5 to generate responses for unknown products.
- **Challenge:** Ensuring accurate voice recognition.
  - **Solution:** Used speech_recognition library with Google Web Speech API for reliable voice input.

Task 3: Flutter App Development

Problem Statement: Develop a simple Flutter application with at least two screens: a home screen and a details screen.

Requirements:

- The home screen should contain a list of items.
- Tapping on an item should navigate the user to the details screen displaying detailed information.
- Implement proper state management.

- Clearly structure your code using the standard project architecture (widgets, screens, and utilities).

Approach:

1. Project Structure:
   - lib/
     - home_screen.dart: Contains the home screen with a list of vehicles.
     - details_screen.dart: Contains the details screen displaying detailed information about a selected vehicle.
     - splash_screen.dart: Contains the splash screen displayed at the start of the app.
     - vehicle_search.dart: Contains the search functionality for filtering vehicles.
2. Home Screen:
   - Displays a grid of vehicle cards.
   - Each card shows an image and name of the vehicle.
   - Tapping on a card navigates to the details screen with detailed information about the selected vehicle.
3. Details Screen:
   - Displays detailed information about the selected vehicle.
   - Includes an image, name, and description of the vehicle.
   - Provides buttons for navigation and actions.
4. Splash Screen:
   - Displays a splash screen with the Yamaha logo for 3 seconds before navigating to the home screen.
5. Search Functionality:
   - Allows users to search for vehicles by name.
   - Filters the list of vehicles based on the search query.

Challenges Faced:

1. **State Management:**
   - Ensuring the state of the vehicle list is properly managed and updated during search operations.
   - Solution: Used a stateful widget and a custom search delegate to handle state changes.
2. **Navigation:**
   - Implementing smooth navigation between screens.
   - Solution: Used Navigator.push and Navigator.pop for screen transitions.
3. **Asset Management:**
   - Ensuring all images are correctly placed in the assets folder and referenced in the code.
   - Solution: Verified the paths and included the assets in pubspec.yaml.

Task 4: Web Development Documentation

Overview

This task involves creating a responsive landing page for a fictional product using React, HTML, CSS, and JavaScript. The landing page includes a navigation bar, main content area, call-to-action button, and is designed to be responsive for both desktop and mobile views.

Approach

1. **Project Structure**:
   - App.js: Main application component that includes the Navbar, Hero, and Footer components.
   - Navbar.js: Component for the navigation bar.
   - Hero.js: Component for the main content area with a call-to-action button.
   - Footer.js: Component for the footer.
   - Corresponding CSS files for styling each component.

2. **Responsive Design**:
    - Used CSS media queries to ensure the layout adapts to different screen sizes.
    - Implemented a hamburger menu for mobile navigation.
    - Adjusted font sizes, padding, and layout for smaller screens.

Components

1. **Navbar Component** (Navbar.js):
    - Contains a logo and a hamburger menu for mobile view.
    - Uses state to toggle the visibility of the navigation links on smaller screens.
2. **Hero Component** (Hero.js):
    - Displays the main content with a welcome message and a call-to-action button.
3. **Footer Component** (Footer.js):
    - Contains a simple footer with copyright information.
4. **App Component** (App.js):
    - Combines all the components to form the complete landing page.

CSS Styling

- **Navbar.css**: Styles for the navigation bar, including responsive adjustments.
- **Hero.css**: Styles for the hero section, ensuring it looks good on all screen sizes.
- **Footer.css**: Styles for the footer, ensuring it stays at the bottom of the page.
- **App.css**: General styles for the application layout.

Challenges and Solutions

- **Responsive Design**: Ensuring the layout adapts well to different screen sizes was challenging. This was addressed by using CSS media queries and testing on various devices.
- **Hamburger Menu**: Implementing a functional and accessible hamburger menu required managing state in React and ensuring proper ARIA attributes for accessibility.