PRAV: Personalized Responsive AI VOICE ASSISTANT

BY

Veerendra amaravathi,srivarsha tandra

1. Abstract

PRAV is a locally-operating AI voice assistant developed in Python using Natural Language Processing (NLP) and deep learning. It recognizes voice or text commands, classifies user intent using a custom-trained neural network, and executes relevant system or web commands. It emphasizes privacy, personalization, and offline availability, differentiating it from cloud-dependent assistants.

2. Problem Statement and Applications

2.1 Problem Statement

Current voice assistants heavily rely on cloud services, compromising user privacy and requiring internet connectivity. They lack flexibility for customization and offline use. PRAV aims to solve these issues by delivering a customizable, privacy-focused assistant that operates locally and adapts to user preferences.

2.2 Applications

- Personal desktop productivity and automation
- Voice-based application and browser control
- Educational assistant for students
- Accessibility aid for people with disabilities
- Smart home device integration (future scope)

3. Workflow Description

- Step 1: User provides voice or typed input.
- Step 2: Voice input is converted to text using speech recognition.
- Step 3: Input text is tokenized and fed to the intent classification model.
- Step 4: The model predicts the user's intent.
- Step 5: Assistant fetches the corresponding response and executes commands.
- Step 6: Response is converted to speech and delivered back to the user.
- Step 7: User preferences and command history are logged for personalization.

4. Modules

| **Module Name** | **Description** | **Technologies Used**
Speech-to-Text Engine	Converts spoken commands to text	SpeechRecognition (Google API)
Intent Classification	Classifies commands into predefined intents	Keras, TensorFlow, sklearn
Text-to-Speech Engine	Converts text responses to speech	pyttsx3
System Integration	Executes OS commands and opens web resources	OS, subprocess
Personalization Module	Stores user preferences and bookmarks	JSON File Handling
Logging Module	Tracks errors and command usage	Python File I/O

5. System Requirements

Software

- Python 3.8+
- TensorFlow, Keras, SpeechRecognition, pyttsx3, nltk, sklearn, numpy

Hardware

- Computer with microphone and speakers
- Minimum 4GB RAM
- Dual-core CPU or better

6. Implementation Details

6.1 Project Structure and Files

- intents.json: Contains user command patterns, intent tags, and response templates.
- train.py: Preprocesses data, trains and saves the intent classification model.
- test.py: Command-line interface for testing intent predictions.
- main.py: The main runtime script handling voice input, intent prediction, command execution, speech output, and personalization.
- utils.py (optional): Utility functions for logging, error handling, and managing user preferences.

6.2 Key Code Snippets

6.2.1 Training Model (train.py)

```
import json
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from sklearn.preprocessing import LabelEncoder
import numpy as np

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

texts, labels = [], []
for intent in data['intents']:
    for pattern in intent['patterns']:
        texts.append(pattern)
        labels.append(intent['tag'])

tokenizer = Tokenizer(num_words=1000)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
max_len = max(len(seq) for seq in sequences)
padded_sequences = pad_sequences(sequences, maxlen=max_len)

encoder = LabelEncoder()
encoded_labels = encoder.fit_transform(labels)

model = Sequential()
model.add(Dense(128, input_shape=(max_len,), activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dense(len(set(labels)), activation='softmax'))

model.compile(loss='sparse_categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
model.fit(padded_sequences, encoded_labels, epochs=1000, batch_size=8)

model.save('intent_model.h5')
import pickle
with open('tokenizer.pkl', 'wb') as f:
    pickle.dump(tokenizer, f)
with open('label_encoder.pkl', 'wb') as f:
```

```
    pickle.dump(encoder, f)
```

6.2.2 Intent Prediction & Command Execution (main.py snippet)

```
import speech_recognition as sr
import pyttsx3
import json
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing.sequence import pad_sequences
import pickle
import os

model = load_model('intent_model.h5')
with open('tokenizer.pkl', 'rb') as f:
    tokenizer = pickle.load(f)
with open('label_encoder.pkl', 'rb') as f:
    encoder = pickle.load(f)

engine = pyttsx3.init()

def listen_command():
    recognizer = sr.Recognizer()
    with sr.Microphone() as source:
        print("Listening...")
        audio = recognizer.listen(source)
    try:
        text = recognizer.recognize_google(audio)
        print(f"User said: {text}")
        return text
    except:
        engine.say("Sorry, I did not catch that.")
        engine.runAndWait()
        return ""

def predict_intent(text):
    sequence = tokenizer.texts_to_sequences([text])
    padded = pad_sequences(sequence, maxlen=30)
    pred = model.predict(padded)
    intent_index = pred.argmax()
    intent_tag = encoder.inverse_transform([intent_index])[0]
    return intent_tag

def execute_command(intent_tag):
    with open('intents.json') as f:
        data = json.load(f)
    for intent in data['intents']:
        if intent['tag'] == intent_tag:
            response = intent['responses'][0]
```

```
            if intent_tag == 'open_browser':
                os.system("start chrome")
            engine.say(response)
            engine.runAndWait()
            break

while True:
    command = listen_command()
    if command:
        intent = predict_intent(command)
        execute_command(intent)
```

6.3 Personalization & Logging

- Stores user preferences (favorite commands, websites) in user_prefs.json.
- Maintains logs.txt to record errors and user interactions for debugging and improvement.
- Supports code explanation requests by fetching explanations from a local knowledge base or APIs.

7. Code Module Summary

Module	Functionality	Key Technologies
train.py	Data loading, preprocessing, training model	Keras, TensorFlow, sklearn
test.py	Command-line model testing	Python CLI
main.py	Voice recognition, intent prediction, command execution, TTS	SpeechRecognition, pyttsx3, OS commands
utils.py	Logging, error handling, personalization	Python File I/O, JSON
intents.json	Dataset of commands, intents, and responses	JSON

8. Development Environment and Tools

- Python 3.8+
- IDE: PyCharm or VSCode
- Libraries: TensorFlow, Keras, SpeechRecognition, pyttsx3, numpy, nltk, sklearn
- OS: Windows 10 (tested)
- Hardware: Microphone, speakers
- Version control: Git

9. Challenges Faced

- Handling noisy or unclear voice input causing recognition errors.
- Balancing model training epochs to optimize accuracy without long delays.
- Ensuring OS command execution compatibility across different platforms.
- Managing ambiguous or unknown commands gracefully to avoid crashes.

10. Project Insights and Findings

- Modular architecture facilitates easy addition and tuning of intents and commands.
- Local execution enhances user privacy and provides offline availability.
- Personalization improves user engagement by saving preferences and frequently used commands.
- Incorporating error handling and logging significantly increases system robustness.
- Future integration with smart home and IoT devices could broaden application scope.

11. Future Scope

- Develop a graphical user interface (GUI) with Tkinter or PyQt.
- Add multilingual voice command support for wider user base.
- Enable cloud synchronization for user data backup and multi-device use.
- Incorporate context-awareness for smarter, conversational interactions.
- Extend assistant capabilities with continuous learning from user behavior.
- Integrate with smart home devices and IoT ecosystems.