import numpy as np

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Embedding, GlobalAveragePooling1D, Dropout, BatchNormalization

from tensorflow.keras.layers import LSTM, Bidirectional, Conv1D, MaxPooling1D

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad\_sequences

from tensorflow.keras.utils import to\_categorical

from tensorflow.keras.regularizers import l2

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics.pairwise import cosine\_similarity

from sklearn.utils.class\_weight import compute\_class\_weight

from sentence\_transformers import SentenceTransformer

import random

import json

import re

from difflib import get\_close\_matches

import os

import tkinter as tk

from tkinter import scrolledtext, font as tkfont, ttk

import threading

import matplotlib.pyplot as plt

from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg

from matplotlib.figure import Figure

import datetime

import time

from itertools import cycle

os.environ['PYTHONHASHSEED'] = '42'

os.environ['TF\_DETERMINISTIC\_OPS'] = '1'

np.random.seed(42)

tf.random.set\_seed(42)

random.seed(42)

with open(r"C:\Users\Administrator\Downloads\Ruchix x Me (IDS).json", 'r', encoding='utf-8') as file:

intents = json.load(file)

semantic\_model = SentenceTransformer('paraphrase-MiniLM-L6-v2')

def preprocess\_text(text):

text = text.lower().strip()

text = re.sub(r'[^\w\s]', '', text)

text = re.sub(r'\s+', ' ', text)

text = re.sub(r'\d+', '', text)

return text

def correct\_input(user\_input, last\_intent=None):

processed\_input = preprocess\_text(user\_input)

if not processed\_input:

return user\_input

all\_patterns = []

if last\_intent and last\_intent in intents:

all\_patterns.extend([preprocess\_text(p) for p in intents[last\_intent]["patterns"]])

all\_patterns.extend([preprocess\_text(p) for data in intents.values() for p in data["patterns"]])

match = get\_close\_matches(processed\_input.strip(), all\_patterns, n=1, cutoff=0.8)

return match[0] if match else user\_input

tokenizer = Tokenizer(oov\_token="<OOV>", filters='')

all\_patterns = []

labels = []

label\_index = {}

for i, (intent, data) in enumerate(intents.items()):

label\_index[intent] = i

for pattern in data["patterns"]:

processed\_pattern = preprocess\_text(pattern)

if processed\_pattern:

all\_patterns.append(processed\_pattern)

labels.append(i)

tokenizer.fit\_on\_texts(all\_patterns)

total\_words = len(tokenizer.word\_index) + 1

sequences = tokenizer.texts\_to\_sequences(all\_patterns)

max\_length = min(50, max(len(seq) for seq in sequences)) if sequences else 10

padded\_sequences = pad\_sequences(sequences, maxlen=max\_length, padding='post', truncating='post')

labels = to\_categorical(labels, num\_classes=len(intents))

X\_train, X\_val, y\_train, y\_val = train\_test\_split(

padded\_sequences,

labels,

test\_size=0.15,

stratify=labels,

random\_state=42

)

y\_train\_classes = np.argmax(y\_train, axis=1)

class\_weights = compute\_class\_weight(

class\_weight='balanced',

classes=np.unique(y\_train\_classes),

y=y\_train\_classes

)

class\_weights = {int(i): float(class\_weights[i]) for i in range(len(class\_weights))}

model = Sequential([

Embedding(total\_words, 128, mask\_zero=True, input\_length=max\_length),

Conv1D(64, 3, activation='relu', padding='same', kernel\_regularizer=l2(0.01)),

MaxPooling1D(2),

Bidirectional(LSTM(64, return\_sequences=True, kernel\_regularizer=l2(0.02))),

Dropout(0.5),

BatchNormalization(),

Bidirectional(LSTM(32, kernel\_regularizer=l2(0.02))),

Dropout(0.4),

BatchNormalization(),

Dense(64, activation='relu', kernel\_regularizer=l2(0.02)),

Dropout(0.4),

Dense(len(intents), activation='softmax')

])

initial\_learning\_rate = 0.001

lr\_schedule = tf.keras.optimizers.schedules.ExponentialDecay(

initial\_learning\_rate,

decay\_steps=800,

decay\_rate=0.85,

staircase=True

)

optimizer = tf.keras.optimizers.Adam(learning\_rate=lr\_schedule, clipnorm=1.0)

model.compile(

loss='categorical\_crossentropy',

optimizer=optimizer,

metrics=['accuracy', tf.keras.metrics.Precision(name='precision'),

tf.keras.metrics.Recall(name='recall'), tf.keras.metrics.AUC(name='auc')]

)

history = model.fit(

X\_train,

y\_train,

epochs=50,

validation\_data=(X\_val, y\_val),

batch\_size=32,

class\_weight=class\_weights,

shuffle=True,

verbose=1

)

intent\_embeddings = {}

for intent, data in intents.items():

patterns = [preprocess\_text(p) for p in data['patterns'] if preprocess\_text(p)]

if patterns:

embeddings = semantic\_model.encode(patterns, batch\_size=32, show\_progress\_bar=False)

intent\_embeddings[intent] = np.mean(embeddings, axis=0)

class ConversationContext:

def \_\_init\_\_(self):

self.last\_intent = None

self.last\_responses = []

self.last\_user\_inputs = []

self.context\_window = 5

self.quest\_keywords = ["quest", "mission", "task", "errand"]

def update(self, intent, response, user\_input):

self.last\_intent = intent

self.last\_responses.append(response)

self.last\_user\_inputs.append(user\_input)

if len(self.last\_responses) > self.context\_window:

self.last\_responses.pop(0)

if len(self.last\_user\_inputs) > self.context\_window:

self.last\_user\_inputs.pop(0)

def get\_context\_bonus(self, intent):

recent\_count = self.last\_responses[-self.context\_window:].count(intent)

bonus = 1.0 + (0.1 \* recent\_count)

if any(keyword in ' '.join(self.last\_user\_inputs[-2:]).lower()

for keyword in self.quest\_keywords):

if "quest" in intent.lower():

bonus \*= 1.5

return min(bonus, 1.5)

conversation\_context = ConversationContext()

def get\_intent(user\_input):

processed\_input = preprocess\_text(user\_input)

if not processed\_input:

return None, 0.0

lower\_input = processed\_input.lower()

if any(greet in lower\_input for greet in ["how are you", "how're you", "hows it going"]):

return "question\_greeting", 0.99

quest\_words = ["quest", "mission", "task", "errand", "activity"]

if any(word in lower\_input for word in quest\_words):

best\_quest\_intent = None

best\_score = 0.0

for intent in ["initiating\_quests", "accepting\_quests", "quest\_progress"]:

if intent in intent\_embeddings:

similarity = cosine\_similarity(

semantic\_model.encode([processed\_input]),

[intent\_embeddings[intent]]

)[0][0]

if similarity > best\_score:

best\_score = similarity

best\_quest\_intent = intent

if best\_quest\_intent and best\_score > 0.5:

return best\_quest\_intent, best\_score

user\_embedding = semantic\_model.encode([processed\_input])

best\_intent = None

best\_score = 0.0

for intent, embedding in intent\_embeddings.items():

if embedding is not None:

similarity = cosine\_similarity(user\_embedding, embedding.reshape(1, -1))[0][0]

similarity \*= conversation\_context.get\_context\_bonus(intent)

if similarity > best\_score:

best\_score = similarity

best\_intent = intent

base\_threshold = 0.7

if best\_intent and "quest" in best\_intent.lower():

base\_threshold = 0.5

word\_count = len(processed\_input.split())

length\_adjustment = min(0.15, word\_count \* 0.01)

threshold = max(base\_threshold - length\_adjustment, 0.5)

return (best\_intent, best\_score) if best\_score > threshold else (None, 0.0)

def get\_intent\_response(user\_input):

if "luna" in user\_input.lower():

phases = ["🌑", "🌒", "🌓", "🌔", "🌕", "🌖", "🌗", "🌘"]

chosen\_phase = random.choice(phases)

return f"{chosen\_phase} love. {chosen\_phase}"

corrected\_input = correct\_input(user\_input, conversation\_context.last\_intent)

intent\_name, confidence = get\_intent(corrected\_input)

current\_time = datetime.datetime.now().strftime("%I:%M %p")

if intent\_name and intent\_name in intents:

response = random.choice(intents[intent\_name]["responses"])

if "{time}" in response:

response = response.format(time=current\_time)

if confidence < 0.85:

similar\_patterns = [

p for p in intents[intent\_name]["patterns"]

if p.lower() in corrected\_input.lower() or

corrected\_input.lower() in p.lower()

]

if similar\_patterns:

response = f"I think you're asking about {intent\_name.replace('\_', ' ')}. {response}"

else:

all\_patterns = [p for data in intents.values() for p in data["patterns"]]

matches = get\_close\_matches(

corrected\_input,

all\_patterns,

n=2,

cutoff=0.6

)

if matches:

response = f"Did you mean something like: '{matches[0]}' or '{matches[1]}'?"

else:

response = random.choice([

"That concept exists in my training, but I didn't quite catch your phrasing.",

"I recognize some keywords but need more context.",

"Try using different phrasing from my training examples."

])

conversation\_context.update(intent\_name, response, user\_input)

return response

class PerformanceGraphs:

def \_\_init\_\_(self, master, history):

self.master = master

self.history = history

master.title("Model Performance Metrics")

master.geometry("800x600")

plt.style.use('ggplot')

self.bg\_color = '#f5f5f5'

self.text\_color = '#333333'

master.configure(bg=self.bg\_color)

self.notebook = ttk.Notebook(master)

self.notebook.pack(fill='both', expand=True)

self.create\_accuracy\_tab()

self.create\_loss\_tab()

def create\_accuracy\_tab(self):

frame = ttk.Frame(self.notebook)

self.notebook.add(frame, text="Accuracy")

fig = Figure(figsize=(8, 4), dpi=100)

ax = fig.add\_subplot(111)

ax.plot(self.history.history['accuracy'], label='Training Accuracy')

ax.plot(self.history.history['val\_accuracy'], label='Validation Accuracy')

ax.set\_title('Model Accuracy Over Epochs')

ax.set\_xlabel('Epoch')

ax.set\_ylabel('Accuracy')

ax.legend()

ax.grid(True)

canvas = FigureCanvasTkAgg(fig, master=frame)

canvas.draw()

canvas.get\_tk\_widget().pack(fill='both', expand=True)

def create\_loss\_tab(self):

frame = ttk.Frame(self.notebook)

self.notebook.add(frame, text="Loss")

fig = Figure(figsize=(8, 4), dpi=100)

ax = fig.add\_subplot(111)

ax.plot(self.history.history['loss'], label='Training Loss')

ax.plot(self.history.history['val\_loss'], label='Validation Loss')

ax.set\_title('Model Loss Over Epochs')

ax.set\_xlabel('Epoch')

ax.set\_ylabel('Loss')

ax.legend()

ax.grid(True)

canvas = FigureCanvasTkAgg(fig, master=frame)

canvas.draw()

canvas.get\_tk\_widget().pack(fill='both', expand=True)

class TypingIndicator(tk.Frame):

def \_\_init\_\_(self, parent):

super().\_\_init\_\_(parent, bg=parent.cget('bg'))

self.dot\_cycle = cycle([". ", ".. ", "..."])

self.label = tk.Label(

self,

text="",

font=('Segoe UI', 10, 'italic'),

fg='#666666',

bg=parent.cget('bg')

)

self.label.pack()

self.animation\_id = None

def start(self, prefix="SR AI is typing"):

self.prefix = prefix

self.\_animate()

def stop(self):

if self.animation\_id:

self.after\_cancel(self.animation\_id)

self.label.config(text="")

def \_animate(self):

dots = next(self.dot\_cycle)

self.label.config(text=f"{self.prefix}{dots}")

self.animation\_id = self.after(300, self.\_animate)

class ChatBubble(tk.Frame):

def \_\_init\_\_(self, parent, text, is\_user=False):

super().\_\_init\_\_(parent, bg=parent.cget('bg'))

self.text = text

self.is\_user = is\_user

bg\_color = parent.cget('bg')

if bg\_color == '#f5f5f5':

bubble\_bg = '#4CAF50' if is\_user else '#E0E0E0'

text\_fg = 'black'

else:

bubble\_bg = '#00aa66' if is\_user else '#333366'

text\_fg = 'white'

self.bubble\_frame = tk.Frame(

self,

bg=bubble\_bg,

padx=12,

pady=8,

highlightthickness=1,

highlightbackground='#888888'

)

self.bubble\_frame.pack(

side='right' if is\_user else 'left',

fill='both',

expand=True

)

self.label = tk.Label(

self.bubble\_frame,

text=text,

wraplength=300,

justify='left',

bg=bubble\_bg,

fg=text\_fg,

font=('Segoe UI', 11)

)

self.label.pack(fill='both', expand=True)

class ChatbotGUI:

def \_\_init\_\_(self, master, history):

self.master = master

self.history = history

master.title("SR AI")

self.dark\_mode = False

self.setup\_ui()

self.master.after(100, self.show\_welcome\_message)

def setup\_ui(self):

self.setup\_styles()

self.main\_frame = tk.Frame(self.master, bg=self.bg\_color)

self.main\_frame.pack(fill='both', expand=True, padx=10, pady=10)

self.create\_header()

self.create\_chat\_area()

self.create\_input\_area()

self.create\_status\_bar()

self.typing\_indicator = TypingIndicator(self.main\_frame)

self.typing\_indicator.pack(fill='x', pady=(0, 5))

def setup\_styles(self):

if self.dark\_mode:

self.bg\_color = '#121212'

self.header\_bg = '#1a1a1a'

self.chat\_bg = '#1e1e1e'

self.input\_bg = '#252525'

self.text\_color = '#ffffff'

self.user\_color = '#00aa66'

self.bot\_color = '#333366'

else:

self.bg\_color = '#f5f5f5'

self.header\_bg = '#ffffff'

self.chat\_bg = '#ffffff'

self.input\_bg = '#ffffff'

self.text\_color = '#333333'

self.user\_color = '#4CAF50'

self.bot\_color = '#E0E0E0'

def create\_header(self):

self.header\_frame = tk.Frame(

self.main\_frame,

bg=self.header\_bg,

padx=10,

pady=10

)

self.header\_frame.pack(fill='x')

self.title\_label = tk.Label(

self.header\_frame,

text="SR AI Assistant",

font=('Segoe UI', 16, 'bold'),

fg=self.text\_color,

bg=self.header\_bg

)

self.title\_label.pack(side='left')

self.theme\_button = tk.Button(

self.header\_frame,

text="🌙" if not self.dark\_mode else "☀️",

command=self.toggle\_theme,

bg=self.header\_bg,

fg=self.text\_color,

bd=0,

relief='flat'

)

self.theme\_button.pack(side='right', padx=5)

self.stats\_button = tk.Button(

self.header\_frame,

text="📊 Stats",

command=self.show\_performance,

bg=self.header\_bg,

fg=self.text\_color,

bd=0,

relief='flat'

)

self.stats\_button.pack(side='right', padx=5)

def create\_chat\_area(self):

self.chat\_frame = tk.Frame(

self.main\_frame,

bg=self.chat\_bg

)

self.chat\_frame.pack(fill='both', expand=True)

self.canvas = tk.Canvas(

self.chat\_frame,

bg=self.chat\_bg,

highlightthickness=0

)

self.scrollbar = ttk.Scrollbar(

self.chat\_frame,

orient='vertical',

command=self.canvas.yview

)

self.canvas.configure(yscrollcommand=self.scrollbar.set)

self.scrollbar.pack(side='right', fill='y')

self.canvas.pack(side='left', fill='both', expand=True)

self.messages\_frame = tk.Frame(

self.canvas,

bg=self.chat\_bg

)

self.messages\_frame.bind(

'<Configure>',

lambda e: self.canvas.configure(

scrollregion=self.canvas.bbox('all')

)

)

self.canvas.create\_window(

(0, 0),

window=self.messages\_frame,

anchor='nw'

)

def create\_input\_area(self):

self.input\_frame = tk.Frame(

self.main\_frame,

bg=self.bg\_color,

padx=5,

pady=5

)

self.input\_frame.pack(fill='x')

self.entry = tk.Entry(

self.input\_frame,

font=('Segoe UI', 12),

bg=self.input\_bg,

fg=self.text\_color,

insertbackground=self.text\_color,

relief='flat',

highlightthickness=1,

highlightbackground='#cccccc',

highlightcolor=self.user\_color

)

self.entry.pack(

side='left',

fill='x',

expand=True,

padx=(0, 5)

)

self.entry.bind('<Return>', self.send\_message)

self.send\_button = tk.Button(

self.input\_frame,

text="Send",

command=self.send\_message,

bg=self.user\_color,

fg='white',

activebackground='#2E7D32',

activeforeground='white',

relief='flat'

)

self.send\_button.pack(side='right')

def create\_status\_bar(self):

self.status\_frame = tk.Frame(

self.main\_frame,

bg=self.bg\_color,

height=20

)

self.status\_frame.pack(fill='x')

self.status\_label = tk.Label(

self.status\_frame,

text="Ready",

font=('Segoe UI', 8),

fg='#666666',

bg=self.bg\_color

)

self.status\_label.pack(side='left', padx=10)

def show\_welcome\_message(self):

self.display\_message(

"AI Assistant",

"Hello! How can I help you today?",

is\_bot=True

)

def toggle\_theme(self):

self.dark\_mode = not self.dark\_mode

self.setup\_styles()

self.update\_theme()

self.theme\_button.config(

text="🌙" if not self.dark\_mode else "☀️"

)

def update\_theme(self):

self.main\_frame.config(bg=self.bg\_color)

self.header\_frame.config(bg=self.header\_bg)

self.chat\_frame.config(bg=self.chat\_bg)

self.input\_frame.config(bg=self.bg\_color)

self.status\_frame.config(bg=self.bg\_color)

self.messages\_frame.config(bg=self.chat\_bg)

self.canvas.config(bg=self.chat\_bg)

self.title\_label.config(

bg=self.header\_bg,

fg=self.text\_color

)

self.theme\_button.config(

bg=self.header\_bg,

fg=self.text\_color

)

self.stats\_button.config(

bg=self.header\_bg,

fg=self.text\_color

)

self.entry.config(

bg=self.input\_bg,

fg=self.text\_color,

insertbackground=self.text\_color,

highlightcolor=self.user\_color

)

self.send\_button.config(

bg=self.user\_color

)

self.status\_label.config(

bg=self.bg\_color,

fg='#666666'

)

self.typing\_indicator.config(bg=self.bg\_color)

self.typing\_indicator.label.config(

bg=self.bg\_color,

fg='#666666'

)

if hasattr(self, 'message\_history'):

for widget in self.messages\_frame.winfo\_children():

widget.destroy()

for sender, message, is\_bot in self.message\_history:

self.display\_message(

sender,

message,

is\_bot,

redraw=True

)

def display\_message(self, sender, message, is\_bot=False, redraw=False):

if not redraw:

if not hasattr(self, 'message\_history'):

self.message\_history = []

self.message\_history.append(

(sender, message, is\_bot))

timestamp = datetime.datetime.now().strftime("%H:%M")

timestamp\_label = tk.Label(

self.messages\_frame,

text=timestamp,

font=('Segoe UI', 8),

fg='#666666',

bg=self.chat\_bg

)

timestamp\_label.pack(

anchor='ne' if not is\_bot else 'nw',

pady=(5, 0)

)

bubble = ChatBubble(

self.messages\_frame,

message,

is\_user=not is\_bot

)

bubble.pack(

fill='x',

padx=10,

pady=(0, 10)

)

self.messages\_frame.update\_idletasks()

self.canvas.yview\_moveto(1.0)

def send\_message(self, event=None):

message = self.entry.get().strip()

if not message:

return

self.display\_message("You", message)

self.entry.delete(0, 'end')

if message.lower() == 'exit':

self.display\_message(

"AI Assistant",

"Goodbye! Have a great day!",

is\_bot=True

)

self.master.after(1500, self.master.destroy)

return

self.typing\_indicator.start()

threading.Thread(

target=self.process\_response,

args=(message,),

daemon=True

).start()

def show\_performance(self):

graphs\_window = tk.Toplevel(self.master)

PerformanceGraphs(graphs\_window, self.history)

def process\_response(self, user\_text):

try:

start\_time = time.time()

response = get\_intent\_response(user\_text)

processing\_time = time.time() - start\_time

min\_delay = max(0.5, len(response) / 30)

if processing\_time < min\_delay:

time.sleep(min\_delay - processing\_time)

finally:

self.master.after(0, self.typing\_indicator.stop)

self.master.after(0, self.display\_message,

"AI Assistant", response, True)

self.master.after(0, lambda: self.status\_label.config(

text=f"Response time: {processing\_time:.2f}s"

))

def run\_gui(history):

root = tk.Tk()

root.geometry("800x600")

ChatbotGUI(root, history)

root.mainloop()

if \_\_name\_\_ == "\_\_main\_\_":

run\_gui(history)