**Package installations**

pip install pandas numpy scikit-learn joblib nltk transformers torch torchvision torchaudio kivy[base] kivy\_examples sounddevice soundfile speechrecognition

python -m nltk.downloader stopwords wordnet

**text\_preprocessing.py**

import pandas as pd  
import re  
import nltk  
from nltk.corpus import stopwords  
from nltk.stem import WordNetLemmatizer  
  
nltk.download('stopwords')  
nltk.download('wordnet')  
  
  
def clean\_text(text):  
 if isinstance(text, str): # Ensure input is a string  
 text = text.lower()  
 text = re.sub(r'[^a-zA-Z\s]', '', text) # Remove special characters & numbers  
 text = re.sub(r'\s+', ' ', text).strip() # Remove extra spaces  
 return text  
 return '' # Return empty string if text is not valid  
  
  
def preprocess\_text(text):  
 if isinstance(text, str):  
 tokens = text.split()  
 stop\_words = set(stopwords.words('english'))  
 tokens = [word for word in tokens if word not in stop\_words]  
 lemmatizer = WordNetLemmatizer()  
 tokens = [lemmatizer.lemmatize(word) for word in tokens]  
 return ' '.join(tokens)  
 return ''  
  
  
def preprocess\_dataset(input\_csv, output\_csv):  
 try:  
 df = pd.read\_csv(input\_csv, encoding='ISO-8859-1') # Ignore unreadable characters  
 if 'TEXT' not in df.columns:  
 raise ValueError("Column 'TEXT' not found in the dataset")  
  
 df['cleaned\_text'] = df['TEXT'].astype(str).apply(clean\_text) # Convert to string before applying functions  
 df['processed\_text'] = df['cleaned\_text'].apply(preprocess\_text)  
  
 df.to\_csv(output\_csv, index=False)  
 print(f"Processed data saved to {output\_csv}")  
 except Exception as e:  
 print(f"Error processing dataset: {e}")  
  
  
# Usage  
preprocess\_dataset('balanced\_call\_dataset.csv', 'preprocessed\_transcripts.csv')

**train\_model.py**

import pandas as pd  
import torch  
from sklearn.model\_selection import train\_test\_split  
from sklearn.preprocessing import LabelEncoder  
from transformers import BertTokenizer, BertForSequenceClassification, Trainer, TrainingArguments  
from sklearn.metrics import classification\_report  
import numpy as np  
import joblib  
  
# Load and prepare data  
df = pd.read\_csv('preprocessed\_transcripts.csv')  
  
# Encode labels  
le = LabelEncoder()  
df['label'] = le.fit\_transform(df['SENTIMENT'])  
  
# Split data  
train\_texts, val\_texts, train\_labels, val\_labels = train\_test\_split(  
 df['processed\_text'].tolist(),  
 df['label'].tolist(),  
 test\_size=0.2,  
 random\_state=42  
)  
  
# Initialize tokenizer  
tokenizer = BertTokenizer.from\_pretrained('bert-base-uncased')  
  
# Tokenize data  
train\_encodings = tokenizer(train\_texts, truncation=True, padding=True, max\_length=128)  
val\_encodings = tokenizer(val\_texts, truncation=True, padding=True, max\_length=128)  
  
# Create dataset class  
class CallDataset(torch.utils.data.Dataset):  
 def \_\_init\_\_(self, encodings, labels):  
 self.encodings = encodings  
 self.labels = labels  
  
 def \_\_getitem\_\_(self, idx):  
 item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}  
 item['labels'] = torch.tensor(self.labels[idx])  
 return item  
  
 def \_\_len\_\_(self):  
 return len(self.labels)  
  
train\_dataset = CallDataset(train\_encodings, train\_labels)  
val\_dataset = CallDataset(val\_encodings, val\_labels)  
  
# Load pre-trained model  
model = BertForSequenceClassification.from\_pretrained('bert-base-uncased', num\_labels=len(le.classes\_))  
  
# Training arguments  
training\_args = TrainingArguments(  
 output\_dir='./results',  
 num\_train\_epochs=5, # Increase epochs  
 per\_device\_train\_batch\_size=16,  
 per\_device\_eval\_batch\_size=32,  
 warmup\_steps=500,  
 weight\_decay=0.01,  
 logging\_dir='./logs',  
 logging\_steps=10,  
 evaluation\_strategy="epoch",  
 learning\_rate=5e-5 # Adjust learning rate  
)  
  
# Train  
trainer = Trainer(  
 model=model,  
 args=training\_args,  
 train\_dataset=train\_dataset,  
 eval\_dataset=val\_dataset  
)  
  
trainer.train()  
  
# Save model and tokenizer  
model.save\_pretrained('./sentiment\_bert')  
tokenizer.save\_pretrained('./sentiment\_bert')  
  
# Save the label encoder  
joblib.dump(le, 'label\_encoder.pkl')  
  
# Evaluation  
predictions = trainer.predict(val\_dataset)  
preds = np.argmax(predictions.predictions, axis=1)  
print(classification\_report(val\_labels, preds, target\_names=le.classes\_))

**predict\_sentiment.py**

import joblib  
import pandas as pd  
from transformers import BertTokenizer, BertForSequenceClassification  
import torch  
  
  
class SentimentAnalyzer:  
 def \_\_init\_\_(self, model\_path, tokenizer\_path, label\_encoder\_path):  
 self.model = BertForSequenceClassification.from\_pretrained(model\_path)  
 self.tokenizer = BertTokenizer.from\_pretrained(tokenizer\_path)  
 self.label\_encoder = joblib.load(label\_encoder\_path)  
  
 def predict(self, text):  
 # Tokenize and prepare input  
 inputs = self.tokenizer(text, return\_tensors='pt', truncation=True, padding=True, max\_length=128)  
  
 # Predict  
 with torch.no\_grad():  
 outputs = self.model(\*\*inputs)  
 logits = outputs.logits  
 predicted\_class = logits.argmax(dim=-1).item()  
  
 # Map class index to sentiment label  
 sentiment\_label = self.label\_encoder.inverse\_transform([predicted\_class])[0]  
 return sentiment\_label  
  
  
# Example usage  
if \_\_name\_\_ == "\_\_main\_\_":  
 analyzer = SentimentAnalyzer('./sentiment\_bert', './sentiment\_bert', 'label\_encoder.pkl')  
  
 test\_texts = [  
 "This service is amazing! I love it!",  
 "I'm very disappointed with your product.",  
 "Can you help me with my account balance?"  
 ]  
  
 for text in test\_texts:  
 sentiment = analyzer.predict(text)  
 print(f"Text: {text}\nPredicted Sentiment: {sentiment}\n")

**app.py**

import os  
import torch  
import joblib  
import csv  
import numpy as np  
import sounddevice as sd  
import soundfile as sf  
import tempfile  
import threading  
import speech\_recognition as sr  
from datetime import datetime  
from transformers import BertTokenizer, BertForSequenceClassification  
from kivy.app import App  
from kivy.uix.boxlayout import BoxLayout  
from kivy.uix.button import Button  
from kivy.uix.label import Label  
from kivy.uix.textinput import TextInput  
from kivy.clock import Clock  
from kivy.uix.popup import Popup  
from kivy.uix.filechooser import FileChooserIconView  
  
  
# Load model components  
MODEL\_PATH = "./sentiment\_bert"  
TOKENIZER\_PATH = "./sentiment\_bert"  
LABEL\_ENCODER\_PATH = "label\_encoder.pkl"  
  
if not all(map(os.path.exists, [MODEL\_PATH, TOKENIZER\_PATH, LABEL\_ENCODER\_PATH])):  
 raise FileNotFoundError("Missing model, tokenizer, or label encoder!")  
  
model = BertForSequenceClassification.from\_pretrained(MODEL\_PATH)  
tokenizer = BertTokenizer.from\_pretrained(TOKENIZER\_PATH)  
label\_encoder = joblib.load(LABEL\_ENCODER\_PATH)  
  
def analyze\_sentiment(text):  
 inputs = tokenizer(text, return\_tensors='pt', truncation=True, padding=True, max\_length=128)  
 with torch.no\_grad():  
 outputs = model(\*\*inputs)  
 predicted\_class = outputs.logits.argmax(dim=-1).item()  
 return label\_encoder.inverse\_transform([predicted\_class])[0]  
  
class SentimentApp(App):  
 def build(self):  
 self.recording = False  
 self.audio\_buffer = []  
 self.samplerate = 44100  
 self.last\_text = ""  
 self.last\_sentiment = ""  
  
 layout = BoxLayout(orientation='vertical', padding=10, spacing=10)  
  
 self.label = Label(text='Call Sentiment Analysis', size\_hint=(1, 0.15))  
 layout.add\_widget(self.label)  
  
 self.text\_input = TextInput(hint\_text='Recognized text will appear here...', size\_hint=(1, 0.15))  
 layout.add\_widget(self.text\_input)  
  
 analyze\_button = Button(text='Analyze Text', size\_hint=(1, 0.1))  
 analyze\_button.bind(on\_press=self.analyze\_text)  
 layout.add\_widget(analyze\_button)  
  
 upload\_button = Button(text='Upload Audio File', size\_hint=(1, 0.1))  
 upload\_button.bind(on\_press=self.choose\_audio\_file)  
 layout.add\_widget(upload\_button)  
  
 self.result\_label = Label(text='Sentiment: ', size\_hint=(1, 0.15))  
 layout.add\_widget(self.result\_label)  
  
 self.toggle\_button = Button(text='Start Recording', size\_hint=(1, 0.1))  
 self.toggle\_button.bind(on\_press=self.toggle\_recording)  
 layout.add\_widget(self.toggle\_button)  
  
 save\_button = Button(text='Save Conversation', size\_hint=(1, 0.1))  
 save\_button.bind(on\_press=self.save\_conversation)  
 layout.add\_widget(save\_button)  
  
 return layout  
  
 def analyze\_text(self, instance):  
 text = self.text\_input.text.strip()  
 if text:  
 sentiment = analyze\_sentiment(text)  
 self.result\_label.text = f'Sentiment: {sentiment}'  
 self.last\_text = text  
 self.last\_sentiment = sentiment  
  
 def toggle\_recording(self, instance):  
 if not self.recording:  
 self.start\_recording()  
 else:  
 self.stop\_recording()  
  
 def start\_recording(self):  
 self.recording = True  
 self.audio\_buffer = []  
 self.toggle\_button.text = "Stop Recording"  
 self.label.text = "Recording... Speak now!"  
 threading.Thread(target=self.\_record\_stream, daemon=True).start()  
  
 def stop\_recording(self):  
 self.recording = False  
 self.toggle\_button.text = "Start Recording"  
 self.label.text = "Processing..."  
  
 def \_record\_stream(self):  
 def callback(indata, frames, time, status):  
 if self.recording:  
 self.audio\_buffer.append(indata.copy())  
 else:  
 raise sd.CallbackStop()  
  
 try:  
 with sd.InputStream(callback=callback, samplerate=self.samplerate, channels=1):  
 while self.recording:  
 sd.sleep(100)  
 except Exception as e:  
 error\_msg = f"Recording error: {str(e)}"  
 Clock.schedule\_once(lambda dt: self.update\_label(error\_msg))  
 return  
  
 full\_audio = np.concatenate(self.audio\_buffer, axis=0)  
 with tempfile.NamedTemporaryFile(delete=False, suffix=".wav") as f:  
 sf.write(f.name, full\_audio, self.samplerate)  
 temp\_audio\_path = f.name  
  
 threading.Thread(target=self.process\_audio\_file, args=(temp\_audio\_path,), daemon=True).start()  
  
 def choose\_audio\_file(self, instance):  
 content = BoxLayout(orientation='vertical')  
 chooser = FileChooserIconView(path=os.path.expanduser("~\\Documents"))  
 content.add\_widget(chooser)  
  
 def load\_selected(instance):  
 selected = chooser.selection  
 if selected:  
 popup.dismiss()  
 threading.Thread(target=self.process\_audio\_file, args=(selected[0],), daemon=True).start()  
  
 load\_btn = Button(text="Load", size\_hint=(1, 0.1))  
 load\_btn.bind(on\_press=load\_selected)  
 content.add\_widget(load\_btn)  
  
 popup = Popup(title="Select Audio File", content=content, size\_hint=(0.9, 0.9))  
 popup.open()  
  
 def process\_audio\_file(self, filepath):  
 recognizer = sr.Recognizer()  
 try:  
 with sr.AudioFile(filepath) as source:  
 audio = recognizer.record(source)  
 text = recognizer.recognize\_google(audio)  
 Clock.schedule\_once(lambda dt: self.update\_text(text))  
 sentiment = analyze\_sentiment(text)  
 Clock.schedule\_once(lambda dt: self.update\_result(sentiment))  
 self.last\_text = text  
 self.last\_sentiment = sentiment  
 except sr.UnknownValueError:  
 Clock.schedule\_once(lambda dt: self.update\_label("Could not understand audio"))  
 except sr.RequestError:  
 Clock.schedule\_once(lambda dt: self.update\_label("Speech recognition service unavailable"))  
 except Exception as e:  
 error\_msg = f"Error: {str(e)}"  
 Clock.schedule\_once(lambda dt: self.update\_label(error\_msg))  
 finally:  
 if os.path.exists(filepath) and "temp" in filepath:  
 os.remove(filepath)  
  
 def update\_label(self, message):  
 self.label.text = message  
  
 def update\_text(self, text):  
 self.text\_input.text = text  
 self.label.text = "Transcription complete."  
  
 def update\_result(self, sentiment):  
 self.result\_label.text = f"Sentiment: {sentiment}"  
  
 def save\_conversation(self, instance):  
 if not self.last\_text or not self.last\_sentiment:  
 self.label.text = "No conversation to save!"  
 return  
  
 save\_path = "conversations.csv"  
 file\_exists = os.path.isfile(save\_path)  
 try:  
 with open(save\_path, mode='a', newline='', encoding='utf-8') as file:  
 writer = csv.writer(file)  
 if not file\_exists:  
 writer.writerow(["Timestamp", "Transcription", "Sentiment"])  
 writer.writerow([datetime.now().isoformat(), self.last\_text, self.last\_sentiment])  
 self.label.text = "Conversation saved successfully!"  
 except Exception as e:  
 self.label.text = f"Save error: {str(e)}"  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 SentimentApp().run()