AI MULTI-MODEL APP

OCTOBER 31 2024

PREPARED BY

SWETHA KUMAR

CAPSTONE PROJECT REPORT

Table of Contents

Table of Contents

EXECUTIVE SUMMARY

The rapid evolution of Artificial Intelligence (AI), Machine Learning (ML), and Deep Learning (DL) has created a dynamic landscape with a broad range of applications. However, these advancements are often fragmented across various platforms, making it challenging for users —especially those new to the field—to gain a holistic understanding and experience with AI technologies. This project addresses this gap by developing a comprehensive, user-centric application that brings diverse AI, ML, and DL models together in one unified platform. The app is designed to cater to both enthusiasts and professionals, offering an accessible, intuitive interface that enables users to explore and engage with different models seamlessly.

By centralizing access to these technologies, the application significantly reduces the time and effort needed to locate, learn, and interact with various models. Users benefit from immediate access to a range of capabilities, from text generation and image recognition to predictive analytics, without the need for extensive technical expertise. This ease of access empowers a broad audience, fostering hands-on experience and facilitating knowledge transfer. Ultimately, the app is designed to democratize AI, promoting greater understanding and interaction with advanced technologies and making AI experimentation both straightforward and enjoyable.

ACKNOWLEDGEMENTS

I am using this opportunity to express my gratitude to everyone who supported me throughout the course of my capstone project. I am thankful for their aspiring guidance, invaluably constructive criticism and friendly advice during the project work. I am sincerely grateful to them for
sharing their truthful and illuminating views on a number of issues related to the project.

Further, I am fortunate to have Mr. ARUL FRANCIS as my mentor. He has readily shared his immense knowledge in data analytics and guided me in a manner that the outcome resulted in enhancing my data skills.

I certify that the work done by me for conceptualizing and completing this project is original and authentic.

CERTIFICATION OF COMPLETION

I certify that the project “ **AI MULTI-MODEL APP** ” was undertaken and completed (OCT 31 2024)

Mentor by: Mr. ARUL FRANCIS
Date : 31st OCT 2024
Place : Chennai

CHAPTER 1

INVENTORY OVERVIEW

In recent years, Artificial Intelligence (AI), Machine Learning (ML), and Deep Learning (DL) have evolved rapidly, impacting industries ranging from healthcare and finance to entertainment and transportation. These technologies leverage complex algorithms and vast datasets to perform tasks that previously required human intelligence, such as object recognition, language processing, and data-driven predictions. However, despite the growing availability of advanced models, their accessibility remains a challenge. Most models are dispersed across different platforms and APIs, often requiring specialized knowledge, tools, and permissions to interact with them effectively. This fragmentation limits the broader audience's ability to explore, experiment, and understand AI capabilities.

CHAPTER 2

OBJECTIVES

The primary objective of this project is to develop a centralized platform that allows users to access and interact with a diverse set of AI, ML, and DL models from a single, cohesive interface. This platform is designed to simplify the process of working with advanced pre-trained models from sources such as Hugging Face and APIs like Google Console and Gemini, enabling users to interact seamlessly with state-of-the-art AI capabilities. Leveraging Streamlit for frontend development and Google Colab for deployment, the app is built to be accessible, efficient, and user-friendly, ensuring that users can interact with these models without extensive setup or technical barriers.

Additionally, the platform incorporates key functionalities, including object detection, text classification, text generation, and multilingual responses, providing a comprehensive suite of AI tools that users can explore and experiment with. Through this project, the aim is to create a versatile and accessible tool that brings cutting-edge AI technology into the hands of a broader audience, fostering greater engagement and understanding of AI-driven applications.

CHAPTER 3

LITERATURE REVIEW

Streamlit for ML and DL Applications

- Streamlit has revolutionized the deployment of machine learning and deep learning applications by providing a simple, interactive, and user-friendly interface.
- It enables rapid prototyping and deployment, allowing data scientists to build and share ML applications without the need for extensive web development expertise.

Role of Hugging Face, Google Console API, and Gemini

- Hugging Face: Known for its extensive repository of pre-trained models in NLP, computer vision, and more, Hugging Face simplifies the integration of these models through user-friendly APIs.
- Hugging Face's model hub offers resources that reduce the need for developers to spend time training models, promoting faster, easier access to advanced AI functionalities.

Gemini API: Delivers multilingual AI capabilities, enhancing applications with language processing features that support diverse user interactions.

- Gemini's multilingual models allow for broader audience engagement and cater to users across different languages, supporting inclusivity in AI applications.

CHAPTER 4

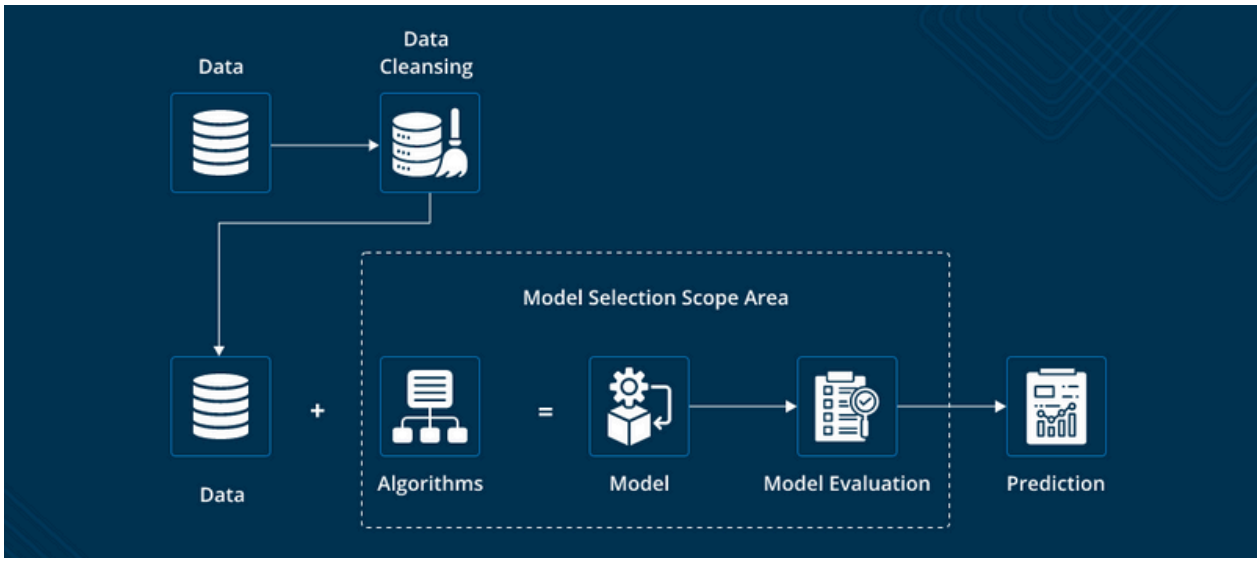
METHODOLOGY

The methodology for this project includes a structured app architecture and careful model integration, both achieved through Streamlit and API-driven functionalities. Streamlit was used to create a user-friendly interface, allowing users to interact with each model through intuitive input and output flows. Each model has a distinct interaction process, such as image upload for object detection and text input for text classification, which seamlessly guides users to the relevant model output.

For model integration, the app incorporates a variety of pre-trained models and APIs from Hugging Face, Google Console, and Gemini. Each model is connected through API calls that retrieve predictions based on user input, which are then displayed in Streamlit's interface.

CHAPTER 5

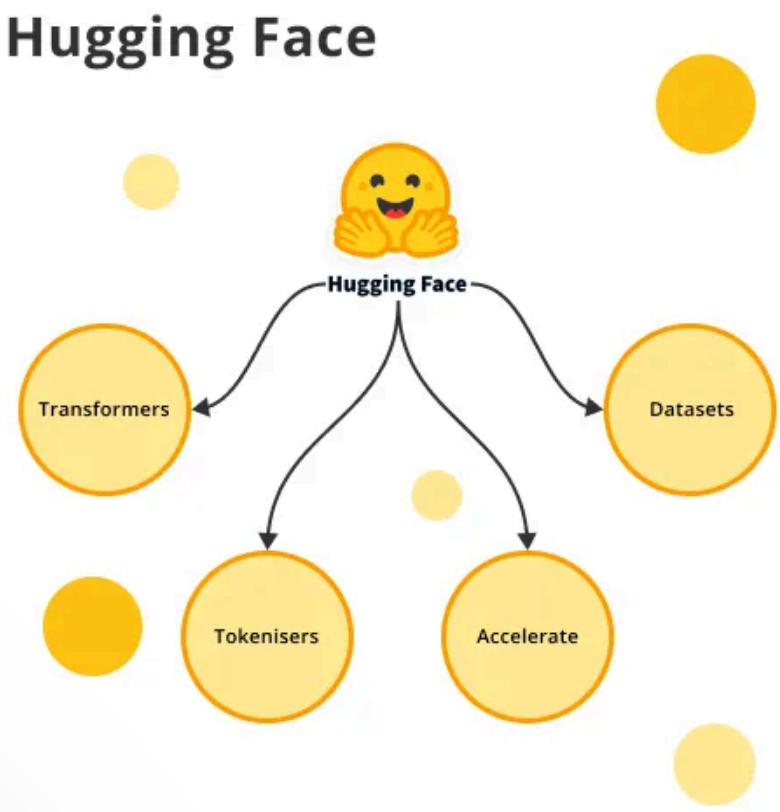
5. RESOURCES:



A unique aspect of Hugging Face is its commitment to accessibility and collaboration. The platform offers a user-friendly API and extensive documentation, making it easier for both beginners and experts to integrate complex AI models into applications without extensive training or setup. Hugging Faces's transformers library, which powers many of its NLP models, widely regarded as one of the most popular machine learning libraries due to its ease of use, versatility, and ability t scale both small-scale and large-scale projects. Additionally, the platform supports model fine-tuning, allowing developers to customize pre-trained models to better fit their specific needs.

5.1 Hugging Face:

Hugging Face is a leading platform in the AI and machine learning community, best known for its comprehensive model hub that provides pre-trained models for various tasks like natural language processing (NLP), computer vision, and beyond. The platform offers resources such as transformers, tokenizers, datasets, and a wide range of APIs that enable developers to easily integrate powerful AI models into their projects. Hugging Face's Transformers library supports a vast selection of models, including BERT, GPT, T5, and more, allowing users to perform tasks like text classification, question answering, text generation, and translation with minimal setup.



5.2 Google Colab:

Google Colab (Colaboratory) is a free, cloud-based platform provided by Google that allows users to write and execute Python code in a Jupyter Notebook environment. With a strong focus on data science, machine learning, and deep learning, Google Colab is particularly popular among students, researchers, and developers who want access to powerful computing resources without requiring dedicated hardware.

Colab provides free access to GPUs (Graphics Processing Units) and TPUs (Tensor Processing Units), which significantly speeds up model training and execution, making it an ideal platform for experimenting with computationally intensive projects. Users can easily install libraries, integrate with cloud storage (like Google Drive), and even connect to other APIs, facilitating streamlined workflows. Moreover, Google Colab supports collaborative editing, enabling multiple users to work on a notebook simultaneously, making it a valuable tool for team projects and shared learning

5.3 Google Console Cloud:

Google Cloud Console is a web-based interface for managing and interacting with Google Cloud services. It provides a centralized platform to access a broad suite of cloud computing resources and tools, enabling users to deploy, monitor, and manage their cloud-based applications and infrastructure. Through the console, developers and organizations can harness Google's powerful cloud offerings, including data storage, machine learning APIs, virtual machines, Kubernetes clusters, and database solutions.

Google Cloud Console is particularly useful for integrating machine learning and AI capabilities, with APIs for natural language processing, vision, translation, and speech recognition. Users can create and manage projects, set up billing, configure permissions, and monitor performance through intuitive dashboards and logging tools. Additionally, it supports programmatic access through APIs, making it easier for developers to integrate Google Cloud services into their applications.

6. ENVIRONMENTAL SETUP

1.Install Streamlit:

```
!pip install streamlit -q
```

2.Install Localtunnel:

```
!npm install -g localtunnel
```

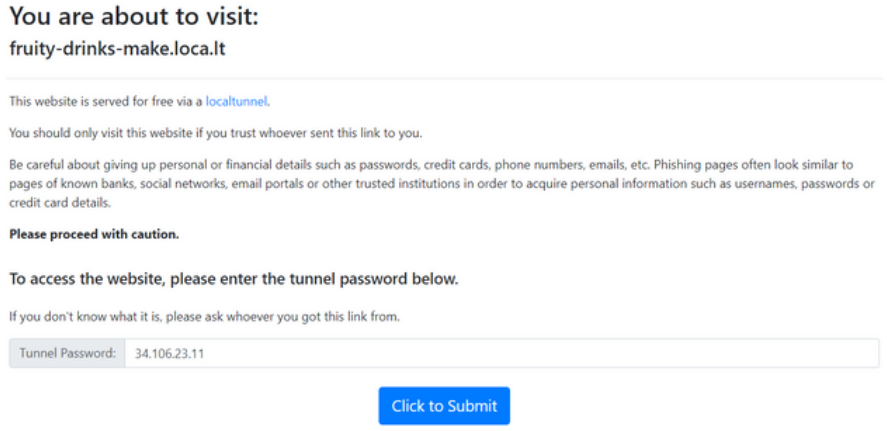
3.Get API Address:

```
!wget -q -O - ipv4.icanhazip.com
```

```
34.75.158.78
```

4.To run Streamlit App:

```
!streamlit run streamlit_app.py & npx localtunnel --port 8501
```



NOTE:

ONCE YOU RUN THE STREAMLIT, YOU WILL BE PROVIDE WITH "YOUR URL" CLICK, THAT NAVIGATE TO OTHER TAB, HERE PASTE YOUR IP ADDRESS. TO RUN THE APP YOUR STREAMLIT CODE NEED TO RUN MUTUALLY ALONG WITH THAT.

6.1 Import and Installation:

```
%%writefile streamlit_app.py

# MAIN
import os
import io
import streamlit as st
import torch

# Neural network library
import torch.nn.functional as F
from transformers import AutoTokenizer, AutoModel , pipeline, AutoModelForTableQuestionAnswering

#PIL (Python Imaging Library): functionality for opening, manipulating, and drawing on images.
from PIL import Image, ImageDraw

#enables sending GET, POST, and other HTTP requests to interact with APIs
import requests

#For regular expression and string manipulation
import re

#API calls to Hugging Face's
from huggingface_hub import InferenceClient

# for interacting with OpenAI's API to access various models
from openai import OpenAI

#to use Google Cloud's Translation API for translating text
from google.cloud import translate_v2 as translate

import pandas as pd
import numpy as np
```

6.2 Basic Requirement:

```
# Set page layout to 'wide'
st.set_page_config(layout="wide", page_title="Multi-Function Streamlit App

# Set up session state to track model loading
if "model_loaded" not in st.session_state:
    st.session_state["model_loaded"] = {
        "similarity": False,
        "audio_classifier": False,
        "table_qa": False,
        "translator" :False
    }

if "first_load" not in st.session_state:
  st.session_state["first_load"] = True
  st.cache_data.clear()  # Clears cached data
  st.cache_resource.clear()  # Clears cached resources (like loaded models
  st.toast("Cache cleared on first load.")

# Initialize session state to store conversation history
if "history" not in st.session_state:
    st.session_state["history"] = []
```

7. MODELS:

1 GEMMA 2(MULTILINGUAL AI)

Gemma 2" is a robust multilingual language model designed to generate output in over 100 international languages. Using Gemini as its backend through an API, Gemma 2 integrates translation support to ensure accurate and fluid responses across diverse languages, allowing it to cater to users from various linguistic backgrounds. The model is tailored for a broad range of applications, including text generation, question answering, and conversational AI. This multilingual support makes it versatile, providing seamless accessibility and usability for a global audience.

Gemma is an innovative multilingual language model designed to facilitate communication and understanding across various languages. It boasts the capability to generate coherent and contextually relevant text in over 100 international languages, making it a valuable tool for diverse applications such as translation, content creation, and customer support.

Source Code:

```
# Gemini AI tab
with tab1:
  try:
    # Streamlit app setup
    st.title("Gemini AI Text Generation")

    json_file_name = '/content/my-project-8754-405207-4d68c3072fa8.json'
    os.environ['GOOGLE_APPLICATION_CREDENTIALS'] = json_file_name
    translate_client = load_translate_client()

    # User input for prompt
    prompt = st.text_area("Enter a prompt for text generation:")

    translate_checkbox = st.checkbox("Translate the generated text")

    if translate_checkbox:
      source_language = st.selectbox("Source Language", options=list(languages.keys()), index=list(languages.keys()).index("English"), key ='selectbox3')
      target_language = st.selectbox("Target Language", options=list(languages.keys()), index=list(languages.keys()).index("Tamil"), key ='selectbox4')

    # Generate text button
    if st.button("Generate Text"):
        with st.spinner("Generating..."):
            generated_text = text_generation(prompt)
            if translate_checkbox:
              generated_text = generated_text.replace('\n\n', '<PARAGRAPH>').replace('\n', '<NEWLINE>')
              translated_text = translator(generated_text, source_language, target_language)
              translated_text = translated_text.replace('<PARAGRAPH>', '  \n\n').replace('<NEWLINE>', '  \n')
        if  translate_checkbox:
          st.write(f"### Generated Text in {target_language}:")
          st.markdown(translated_text, unsafe_allow_html=True)
        else:
          st.write(f"### Generated Text:")
          st.markdown(generated_text)

  except Exception as e:
    st.error(f"An error occurred: {e}")
```

Pipeline:

1 Streamlit App Setup:

- The app is titled "Gemini AI Text Generation".
- It uses Google Cloud for language translation, as shown by setting the GOOGLE_APPLICATION_CREDENTIALS environment variable to a JSON file for authentication.
- A translation client (translate_client) is loaded for translation operations.

2 User Input for Prompt:

- A text area is created where users can enter a prompt for text generation.
- There is a checkbox (translate_checkbox) that allows users to choose if they want the generated text to be translated.

3 Translation Options:

- If translation is enabled, the user selects the source language (default is "English") and target language (default is "Tamil") using two dropdown selectors.

4 Generate Text Button:

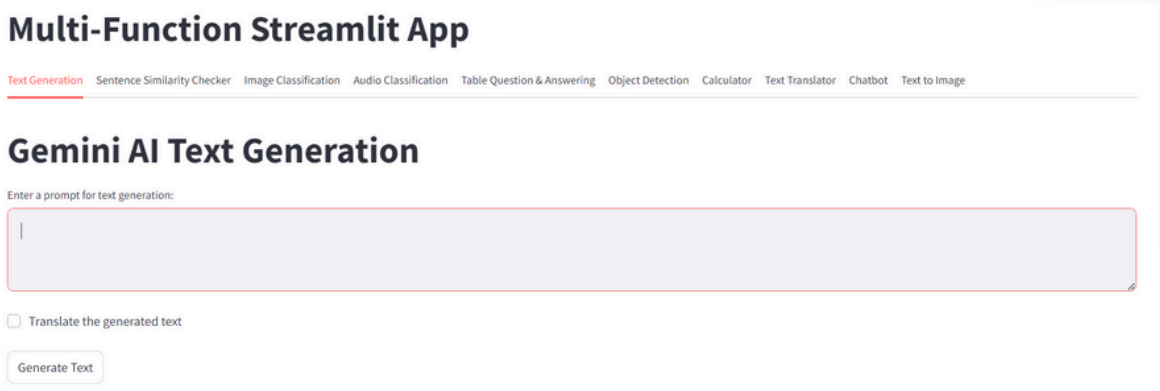
- When the "Generate Text" button is pressed, the app starts generating text.
- If translation is requested:
- The generated text is preprocessed to replace newline characters (\n) with custom tags (<PARAGRAPH> and <NEWLINE>).
- This text is then sent to the translator with the chosen source and target languages.
- After translation, the placeholders are converted back to newlines for formatting.
- The generated or translated text is displayed in markdown format. The translated text header includes the target language.

5 Error Handling:

- If any exception occurs, an error message is displayed.

Output Snippet:

1



2

Translated Text:

செயற்கை நுண்ணறிவு (AI)

வரையறை:

செயற்கை நுண்ணறிவு (AI) என்பது இயந்திரங்கள், குறிப்பாக கணினி அமைப்புகள் மூலம் மனித நுண்ணறிவு செயல்முறைகளை உருவகப்படுத்துவதைக் குறிக்கிறது. கணினிகள் சிந்திக்கவும், கற்றுக்கொள்ளவும், பகுத்தறிவு செய்யவும் மற்றும் தாங்களாகவே முடிவெடுக்கும் திறனை இது உள்ளடக்கியது.

முக்கிய கருத்துக்கள்:

1. இயந்திர கற்றல் (ML):

- கணினிகளுக்கு வடிவங்களை அடையாளம் காணவும், அறிவை ஊகிக்கவும், தரவுகளின் அடிப்படையில் கணிப்புகளைச் செய்யவும் பயிற்சி அளிக்கிறது.
- துணை வகைகளில் மேற்பார்வையிடப்பட்ட கற்றல், மேற்பார்வை செய்யப்படாத கற்றல் மற்றும் வலுவூட்டல் கற்றல் ஆகியவை அடங்கும்.

3

Source Language

English

Target Language

Arabic

Generate Text

Generated Text in Arabic:

أباتشي سبارك

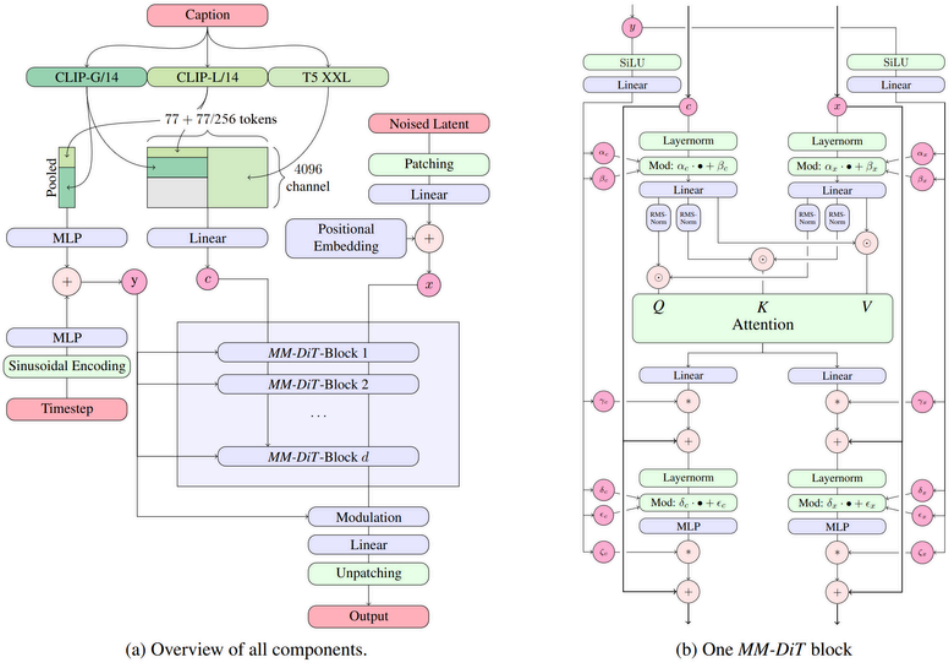
تعريف:

Apache Spark هو إطار عمل للحوسبة الموزعة يتيح معالجة البيانات بسرعة وكفاءة على مجموعات البيانات الضخمة. وهو محرك مفتوح المصدر يعمل على نظام Hadoop الموزعة الأخرى أو أنظمة الملفات الموزعة (DFS).

الميزات الرئيسية:

- **السرعة:** يستخدم Spark نموذج الحوسبة في الذاكرة الذي يقلل بشكل كبير من وقت المعالجة مقارنة بأنظمة معالجة الدفعات التقليدية.
- **قابلية التوسع:** يمكن لـ Spark التعامل مع مجموعات بيانات ضخمة موزعة على خوادم متعددة، مما يتيح معالجة التيرابايتات إلى البيتابايتات من البيانات.
- **التسامح مع الأخطاء:** يوفر Spark آليات مدمجة للتسامح مع الأخطاء لضمان تقليل فقد البيانات وفشل النظام.
- **تجريدات البيانات:** يوفر Spark مجموعة غنية من تجريدات البيانات، بما في ذلك RDDs (مجموعات البيانات الموزعة المرنة)، وDataFrames، وجداول SQL، مما يجعل من السهل العمل مع البيانات واسعة النطاق.
- **واجهات برمجة التطبيقات:** يدعم Spark واجهات برمجة تطبيقات متعددة، بما في ذلك Scala، وJava، وPython، وR، مما يوفر المرونة في معالجة البيانات.

2 TEXT TO IMAGE



(a) Overview of all components.

(b) One *MM-DiT* block

Multimodal Diffusion Transformer (MMDiT) text-to-image model that features improved performance in image quality, typography, complex prompt understanding, and resource-efficiency.

Text-to-image generation is a field within artificial intelligence and deep learning where models generate images based on natural language descriptions. This type of model translates a user-provided textual prompt into a visual representation, capturing objects, scenes, and styles described in the text. Text-to-image technology has numerous applications, from creating concept art to designing personalized visuals for marketing, social media, or virtual worlds.

Source Code:

```python
# Text to Image
with tab10:
  try:
    # Streamlit UI
    st.title("Text-to-Image Generation App")
    st.write("Generate images from text prompts using Stable Diffusion 3.5 (powered by Hugging Face API).")

    # Text input for the prompt
    prompt = st.text_input("Enter a description (e.g., 'Astronaut riding a horse')")

    # Generate button
    if st.button("Generate Image"):
        with st.spinner("Generating image..."):
            generated_image = text_to_image(prompt)

            if generated_image:
                st.image(generated_image, caption="Generated Image", use_column_width=True)
            else:
                st.error("Failed to generate image.")
  except Exception as e:
    st.error(f"An error occurred: {e}")
```

```python
# Text to Image
def text_to_image(prompt):
  # Define the Hugging Face API endpoint and headers with your API key
  API_URL = "https://api-inference.huggingface.co/models/stabilityai/stable-diffusion-3.5-large"
  headers = {"Authorization": "Bearer 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"}  # Replace with your actual token

  # Function to call the Hugging Face model API
  def generate_image(prompt):
      payload = {"inputs": prompt}
      response = requests.post(API_URL, headers=headers, json=payload)

      if response.status_code == 200:
          # Process the image content
          image = Image.open(io.BytesIO(response.content))
          return image
      else:
          st.error(f"Error: {response.status_code} - {response.text}")
          return None

  return generate_image(prompt)
```

Pipeline:

1 User Interface Setup (Streamlit):

- The Streamlit UI is initialized with a title, description, and text input box for entering a prompt (e.g., "Astronaut riding a horse").
- This UI component allows the user to provide a description of the image they want to generate.

2 User Prompt Collection:

- The prompt entered by the user is captured and passed into a function for further processing.
-

3 API Configuration:

- The code defines the API endpoint (API_URL) for the Stable Diffusion model hosted on the Hugging Face platform.
- An authorization header with an API key is set up to authenticate requests to the API.

4 Generate Image Function:

- A function named generate_image is defined to call the Hugging Face API.
- The function:
  - Accepts the user-provided prompt as input.
  - Sends a POST request with the prompt to the API endpoint.
  - Checks if the response is successful (status code 200).
  - If successful, it decodes the response and extracts the image content.
  - If unsuccessful, it displays an error message with the response status.

5 Image Generation Trigger:

- The user can click a "Generate Image" button to start the image generation process.
- When clicked, a loading spinner ("Generating image...") is displayed to indicate processing.

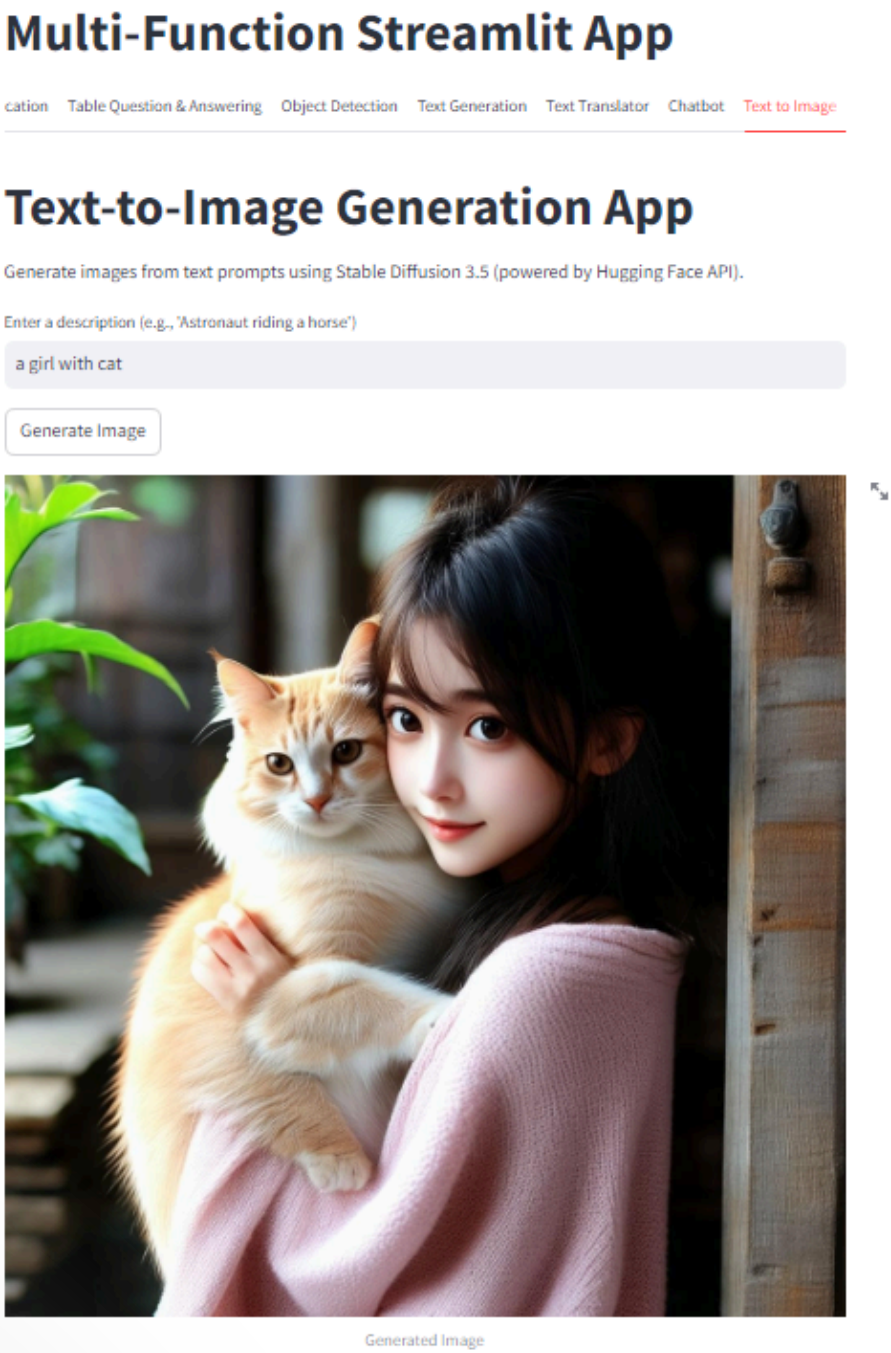
6 Display Generated Image:

- If the image is successfully generated, it is displayed on the Streamlit app with a caption ("Generated Image").
- If image generation fails, an error message is shown to the user.

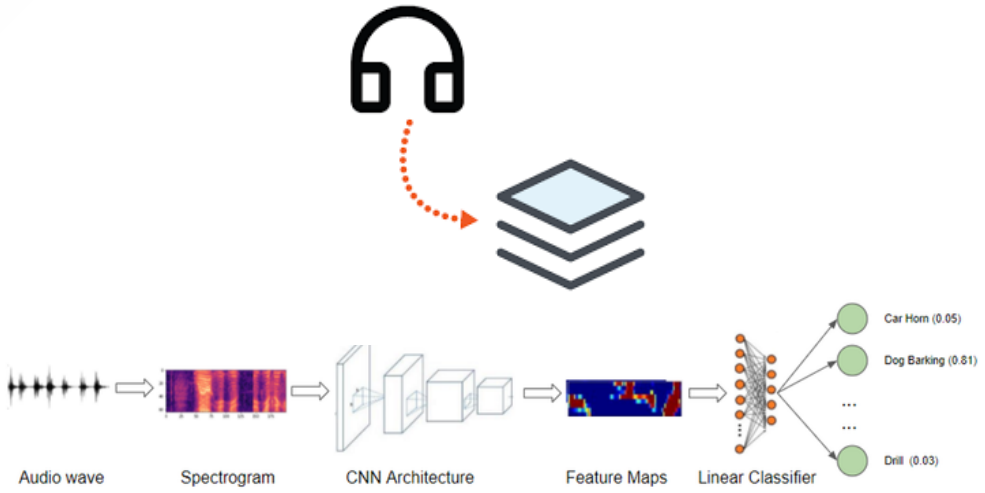
7 Error Handling:

- If any exceptions occur during the process, they are caught and displayed with an error message.

Output Snippet:



3 AUDIO CLASSIFICATION



Audio Spectrogram Transformer (AST) model fine-tuned on AudioSet. It was introduced in the paper AST: Audio Spectrogram Transformer by Gong et al. and first released in this repository.The Audio Spectrogram Transformer is equivalent to ViT, but applied on audio. Audio is first turned into an image (as a spectrogram), after which a Vision Transformer is applied. The model gets state-of-the-art results on several audio classification benchmarks.

Audio Classification is a process in machine learning where a model categorizes audio signals into predefined classes. These classes could represent various categories, like genres in music, types of sounds (e.g., speech, laughter, animal sounds), or specific spoken words in speech recognition. Audio classification models are used in diverse applications, such as voice assistants, speech recognition, music genre classification, environmental sound recognition, and medical diagnosis based on audio signals.

Source Code:

```python
#Audio Classification:
def audio_classifier(audio,audio_pipe):

  def classify_audio(audio):
      """
      Classifies an audio file and returns the top prediction.

      Parameters:
          audio_path (str): File path of the audio file provided by Streamlit.

      Returns:
          str: Label of the predicted class.
      """
      # Perform classification
      predictions = pipe(audio)

      # Format predictions for output
      return predictions[0]['label']

  pipe = audio_pipe

  return classify_audio(audio)
```

```python
# Audio Classification:
with tab4:
  try:
    st.title("Audio Classification with AST Model")

    with st.spinner("Loading model..."):
        audio_pipe = load_audio_classifier()

    st.markdown("Upload an audio file to get predictions.")

    # Set up audio input
    audio_input = st.file_uploader("Upload Audio", type=["mp3", "wav", "ogg"])

    if audio_input is not None:
        # Save the uploaded file temporarily
        audio_path = f"temp_audio.{audio_input.name.split('.')[-1]}"
        with open(audio_path, "wb") as f:
            f.write(audio_input.read())

        # Add a button to classify the audio
        if st.button("Classify Audio"):
            with st.spinner("Classifying..."):
                prediction = audio_classifier(audio_path, audio_pipe)
                st.success(f"Predicted Label: {prediction}")
  except Exception as e:
    st.error(f"An error occurred: {e}")
```

Pipeline:

1 Set Up User Interface in Streamlit (tab4):

- The Streamlit app initializes in a specific tab (tab4) with a title, "Audio Classification with AST Model."
- An instruction is provided for the user to upload an audio file to receive classification predictions.

2 Model Loading:

- Inside a spinner (loading indicator), the function load_audio_classifier() is called to load the pre-trained AST (Audio Spectrogram Transformer) model, which will be used for classification.
- The loaded model is assigned to the variable audio_pipe, making it accessible for audio classification.

3 Audio File Upload:

- The user is prompted to upload an audio file in one of the supported formats (mp3, wav, or ogg) via the Streamlit file uploader.

4 Audio Classification Button:

- Once an audio file is uploaded, a "Classify Audio" button appears.
- When the button is clicked, the audio_classifier function is called to classify the audio file.

5 Audio Classification Process (audio_classifier function):

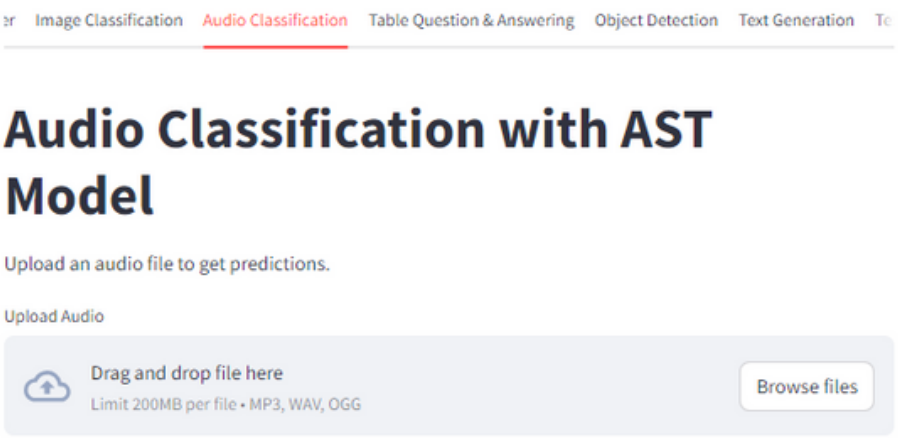
- The audio_classifier function defines a nested helper function classify_audio which:
- Accepts an audio file path as input.
- Uses the preloaded audio_pipe model to generate predictions on the audio.
- Retrieves and returns the label of the top prediction.
- The audio_classifier function then calls classify_audio with the provided audio path and returns the predicted label to the Streamlit app.

6 Display Prediction:
The predicted label is displayed on the app interface as "Predicted Label: <label>," informing the user of the audio classification result.

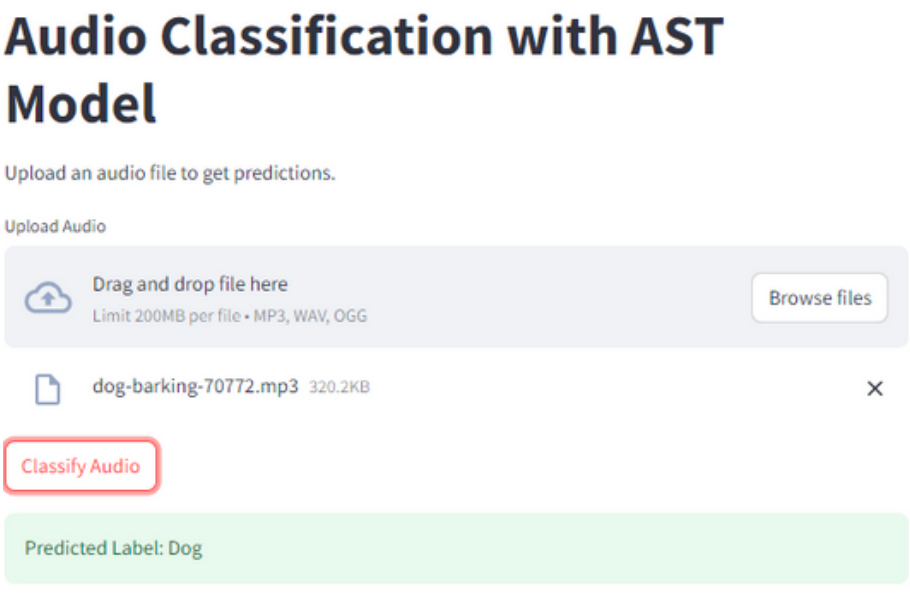
7 Error Handling:
If any error occurs during the workflow, it is caught by the try-except block, and an error message is displayed to the user with the specific error details.

Output Snippets:

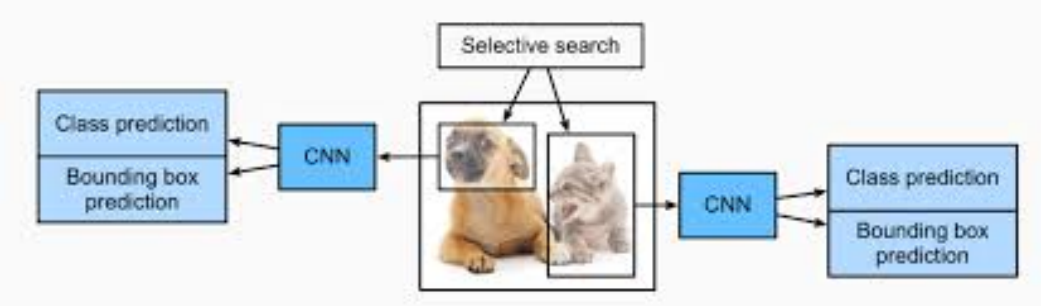
1



2



4 OBJECT DETECTION



DEtection TRansformer (DETR) model trained end-to-end on COCO 2017 object detection (118k annotated images). It was introduced in the paper End-to-End Object Detection with Transformers by Carion et al. and first released in this repository.

The DETR model is an encoder-decoder transformer with a convolutional backbone. Two heads are added on top of the decoder outputs in order to perform object detection: a linear layer for the class labels and a MLP (multi-layer perceptron) for the bounding boxes. The model uses so-called object queries to detect objects in an image. Each object query looks for a particular object in the image.

The model is trained using a "bipartite matching loss": one compares the predicted classes + bounding boxes of each of the N = 100 object queries to the ground truth annotations, padded up to the same length N (so if an image only contains 4 objects, 96 annotations will just have a "no object" as class and "no bounding box" as bounding box).

Source Code:

```python
#Object Detection:
def object_detection(image_data):
  # Hugging Face API details
  API_URL = "https://api-inference.huggingface.co/models/facebook/detr-resnet-101"
  headers = {"Authorization": "Bearer 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"}

  def query(image_data):
      """Sends image data to the Hugging Face API for object detection."""
      response = requests.post(API_URL, headers=headers, data=image_data)
      if response.status_code == 200:
          return response.json()  # Successful response
      elif response.status_code == 503:
          st.warning("Model is still loading. Retrying...")
          time.sleep(20)
      else:
          st.error(f"Error: {response.status_code} - {response.text}")
          return None

  def detect_objects(image):
      """Detects objects in the given image using the Hugging Face API."""
      # Save the uploaded image temporarily
      image.save("temp_image.webp")

      # Read the saved image
      with open("temp_image.webp", "rb") as f:
          image_data = f.read()

      output = query(image_data)
      return output
  return detect_objects(image_data)
```

```python
# Object Detection:
with tab6:
  try:
    # Streamlit app layout
    st.title("Object Detection with DETR Model")

    def draw_boxes(image, boxes, scores, labels, threshold=0.5):
      draw = ImageDraw.Draw(image)
      for i, (box, score) in enumerate(zip(boxes, scores)):
          if score >= threshold:
              xmin, ymin, xmax, ymax = box
              draw.rectangle([xmin, ymin, xmax, ymax], outline="red", width=3)
              draw.text((xmin, ymin), f"{labels[i]}: {score:.2f}", fill="red")
      return image

    uploaded_file = st.file_uploader("Upload an image...", type=["jpg", "jpeg", "png","webp"])

    if uploaded_file is not None:
        image = Image.open(uploaded_file)
        st.image(image, caption='Uploaded Image', width = 300)

        if st.button("Detect Objects"):
            with st.spinner("Detecting objects..."):
                results = object_detection(image)

                if results and isinstance(results, list):
                    boxes, scores, labels = [], [], []
                    for obj in results:
                        if "box" in obj and "score" in obj and "label" in obj:
                          if obj["score"] >= 0.7:
                            box = obj["box"]
                            score = obj["score"]
                            label = obj["label"]
                            boxes.append([box["xmin"], box["ymin"], box["xmax"], box["ymax"]])
                            scores.append(score)
                            labels.append(label)

                    # Draw boxes on the image
                    image_with_boxes = draw_boxes(image, boxes, scores, labels)
                    st.image(image_with_boxes, caption='Detected Objects', use_column_width=True)

                    st.write("### Detected Objects:")
                    for label, score in zip(labels, scores):
                        if score >= 0.7:
                            st.write(f"**Label:** {label}, **Score:** {score:.2f}")
                else:
                    st.error("Unexpected API response format. Please check the response structure.")
```

Pipeline:

1 Set Up User Interface in Streamlit (tab6):

- The Streamlit app initializes in a specific tab (tab6) with a title, "Object Detection with DETR Model."
- Users are prompted to upload an image file (jpg, jpeg, png, or webp format) using the Streamlit file uploader.

2 Define draw_boxes Function:

- A helper function, draw_boxes, is defined to draw bounding boxes and labels around detected objects in the image.
- For each detected object (based on a score threshold of 0.5 by default), a red rectangle is drawn around the object, and its label and confidence score are displayed at the top-left corner of the bounding box.

3 Upload Image File:

- Once the user uploads an image, it is displayed in the Streamlit app with the caption "Uploaded Image" and a set width of 300 pixels.
- A "Detect Objects" button appears to initiate the object detection process.

4 Object Detection Button:

- When the "Detect Objects" button is clicked, the object_detection function is called with the uploaded image as input.
- A spinner appears, indicating that object detection is in progress.

5 Object Detection Process (object_detection function):

- The object_detection function (assumed to use the DETR model) returns a list of detected objects, where each object contains:
- box: Coordinates of the bounding box (xmin, ymin, xmax, ymax).
- score: Confidence score of the detection.
- label: Detected class label of the object.
- A threshold score of 0.7 is applied to filter out detections with low confidence.

Output Snippet:

1

Multi-Function Streamlit App

e Classification | Audio Classification | Table Question & Answering | Object Detection | Text Generation | Text Transla

Object Detection with DETR Model

Upload an image...

Drag and drop file here

Limit 200MB per file • JPG, JPEG, PNG

Browse files

Detect Objects

2

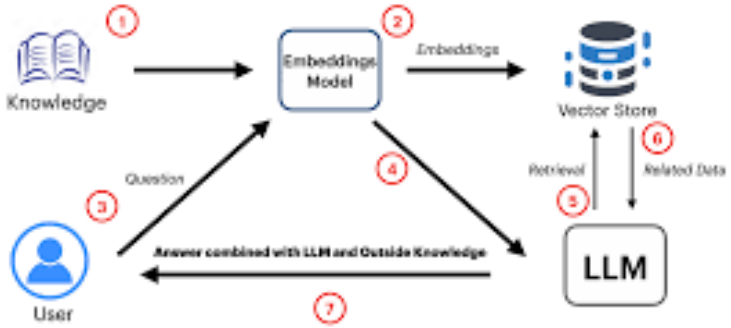
Detected Objects

Detected Objects:

Label: zebra, **Score:** 0.74

Label: giraffe, **Score:** 0.83

5 CHATBOT-TEXT GENERATION



A text generation chatbot is an AI-powered conversational agent that generates responses based on user input using natural language processing and machine learning models. It interprets the user's text prompt, processes it through a language model (often fine-tuned on large text datasets), and produces relevant, coherent replies. This type of chatbot can simulate human-like conversations, answer questions, provide recommendations, and engage in interactive storytelling. Text generation chatbots are commonly implemented using transformer-based models like GPT or BERT and can be customized to adapt to specific tones, languages, or subject areas to create engaging and personalized user experiences.

Source Code:

```
# Chatbot response function
def get_chatbot_response(user_message, history):
    """
    Generates a response for the chatbot interface.

    Parameters:
        user_message (str): The latest user message.
        history (list): Chat history containing previous interactions.

    Returns:
        list: Updated history including the model's response.
    """
    # Prepare the conversation history for the model
    messages = [{"role": "user", "content": user_message}]

    # Stream model response
    response_content = ""

    try:
        # Chat completions with streaming
        stream = client.chat.completions.create(
            model="mistralai/Mistral-Nemo-Instruct-2407",
            messages=messages,
            max_tokens=500,
            stream=True
        )

        # Accumulate response content from streamed chunks
        for chunk in stream:
            response_content += chunk.choices[0].delta.content

        # Append user message and model response to history
        history.append((user_message, response_content))

    except Exception as e:
        response_content = f"An error occurred: {str(e)}"
        history.append((user_message, response_content))

    return history

return get_chatbot_response(user_message, st.session_state.history)
```

```
# Chatbot
with tab9:
  try:
    # Initialize Streamlit app layout
    st.title("Chatbot with Mistral-Nemo Model")
    st.markdown("This is a chatbot interface powered by the Mistral-Nemo model.")

    # Temporary variable to hold the input message
    if "temp_input" not in st.session_state:
        st.session_state.temp_input = ""

    # Input field for user message
    user_message = st.text_input("Ask anything...", key="input_message")

    # Handle user message submission
    if st.button("Send"):
        if user_message:
            # Call chatbot function
            history = chatbot(user_message)

            # Clear the temporary input state after sending
            st.session_state.temp_input = ""

            # Display chat history
            st.write("### Chat History")
            for user_msg, bot_resp in history:
                st.write(f"**User:** {user_msg}")
                st.write(f"**Bot:** {bot_resp}")
        else:
            st.warning("Please enter a message.")

  except Exception as e:
    st.error(f"An error occurred: {e}")
```

Pipeline:

- **Imports:** Import necessary libraries, including Streamlit and the OpenAI client for interacting with the Mistral-Nemo model.

- **OpenAI Client Initialization:** Set up the OpenAI API client with your API key.

- **Chatbot Response Function:** Define the get_chatbot_response function that:
- Takes the user message and chat history as input.
- Prepares the messages for the model and streams the response.
- Accumulates the response from streamed chunks and appends both the user message and the model's response to the history.
- Handles exceptions and appends any error messages to the history.

- **Streamlit Layout:** Initialize the Streamlit app layout with a title and description.

- **Session State:** Initialize session state to hold the chat history if it doesn't exist.

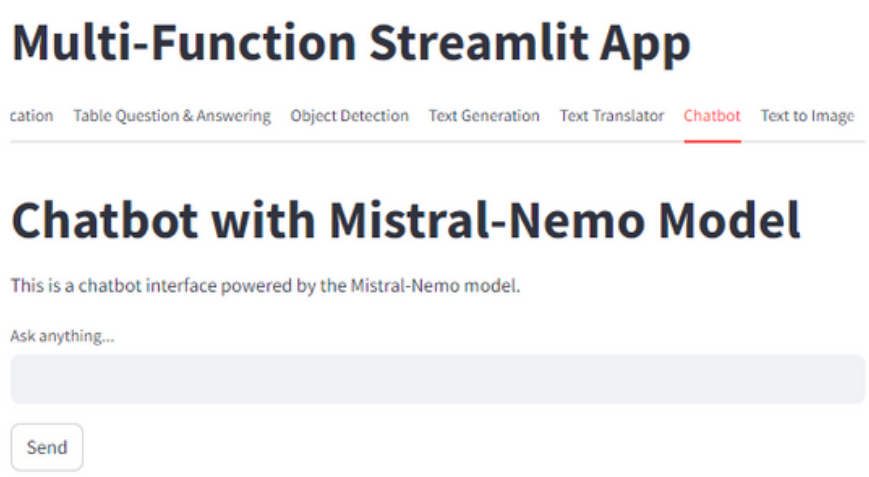
- **User Input:** Create a text input field for user messages.

- **Message Submission Handling:** When the "Send" button is clicked:
- If a message is present, call get_chatbot_response to get the updated history.
- Clear the input field after submission.
- Display the updated chat history.

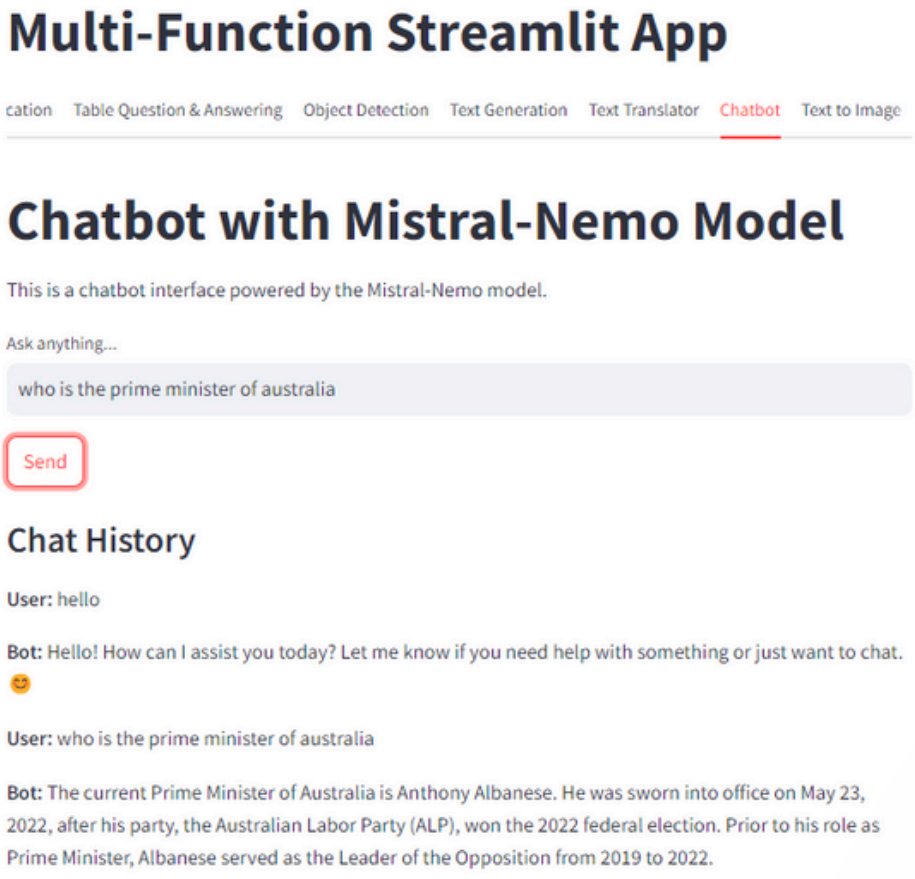
- **Display History on Load:** Display the chat history when the app is loaded.

Output Snippets:

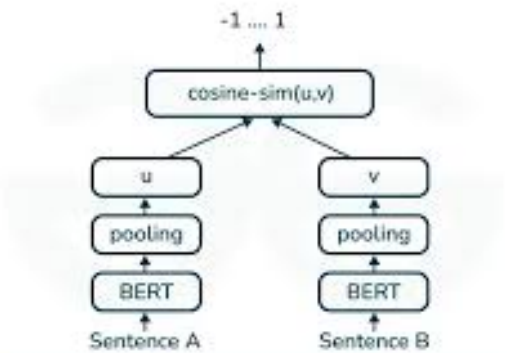
1



2



6 SENTENCE SIMILARITY



Sentence similarity refers to the degree of equivalence or likeness between two sentences in terms of meaning, context, or structure. It plays a crucial role in various natural language processing (NLP) tasks, such as information retrieval, machine translation, text summarization, and semantic search.

Syntactic Similarity:

- This involves comparing the grammatical structure and arrangement of the sentences. Sentences can be syntactically similar even if they differ semantically. For example, "She loves ice cream" and "Ice cream loves her" share a similar structure but have opposite meanings.

Cosine Similarity:

- A common mathematical approach to measure similarity between two vectors (such as word embeddings) is cosine similarity. It calculates the cosine of the angle between the vectors, yielding a value between -1 and 1, where 1 indicates identical sentences, 0 indicates orthogonality (no similarity), and -1 indicates completely opposite meanings.

Jaccard Similarity:

- This method measures the similarity between two sets by comparing the size of their intersection to the size of their union. For sentences, this could involve tokenizing the sentences into words and calculating the ratio of shared words to total unique words.

Source Code:

```python
def similarity(source_sentence, comparison_sentences,sim_tokenizer, sim_model):

    # Mean Pooling - Take attention mask into account for correct averaging
    def mean_pooling(model_output, attention_mask):
        token_embeddings = model_output[0]
        input_mask_expanded = attention_mask.unsqueeze(-1).expand(token_embeddings.size()).float()
        return torch.sum(token_embeddings * input_mask_expanded, 1) / torch.clamp(input_mask_expanded.sum(1), min=1e-9)

    def get_similarity(source_sentence, comparison_sentences):
        sentences = [source_sentence] + comparison_sentences
        encoded_input = sim_tokenizer(sentences, padding=True, truncation=True, return_tensors='pt')

        with torch.no_grad():
            model_output = sim_model(**encoded_input)
        embeddings = mean_pooling(model_output, encoded_input['attention_mask'])
        embeddings = F.normalize(embeddings, p=2, dim=1)
        source_embedding = embeddings[0]
        similarity_scores = F.cosine_similarity(source_embedding, embeddings[1:]).tolist()

        results = [[comparison_sentences[i], {similarity_scores[i]}] for i in range(len(comparison_sentences))]
        results.sort( key=lambda x: x[1], reverse=True)
        results = [[comparison_sentences[i], f"{round(similarity_scores[i] * 100, 2)}%"] for i in range(len(comparison_sentences))]
        return results

    return get_similarity(source_sentence, comparison_sentences)
```

```python
# Sentence Similarity Checker tab
with tab2:
  try:
    st.header("Sentence Similarity Checker")

    with st.spinner("Loading model..."):
            sim_tokenizer, sim_model = load_similarity_model()

    with st.form("similarity_form"):
      source_sentence = st.text_area("Source Sentence", placeholder="Enter the source sentence...")
      comparison_sentences_input = st.text_area("Sentences to Compare", placeholder="Enter sentences to compare, separated by new lines")
      if st.form_submit_button("Check Similarity"):
          if source_sentence and comparison_sentences_input:
            with st.spinner("Finding Similarities..."):
              comparison_sentences = comparison_sentences_input.splitlines()
              similarity_results = similarity(source_sentence, comparison_sentences,sim_tokenizer, sim_model)
              st.write("### Similarity Scores:")
              for sentence, score in similarity_results:
                  st.write(f"**{sentence}**: {score}")
          else:
              st.warning("Please enter both source and comparison sentences.")

  except Exception as e:
    st.error(f"An error occurred: {e}")
```

Pipeline:

1 Import Required Libraries:

- The code imports necessary libraries, including Streamlit for the user interface, Torch for tensor operations, and Hugging Face's Transformers for the pre-trained model and tokenizer.

2 **Load Similarity Model:**

- The load_similarity_model function initializes the tokenizer and model from the Mistral-Nemo repository.

3 Calculate Similarity:

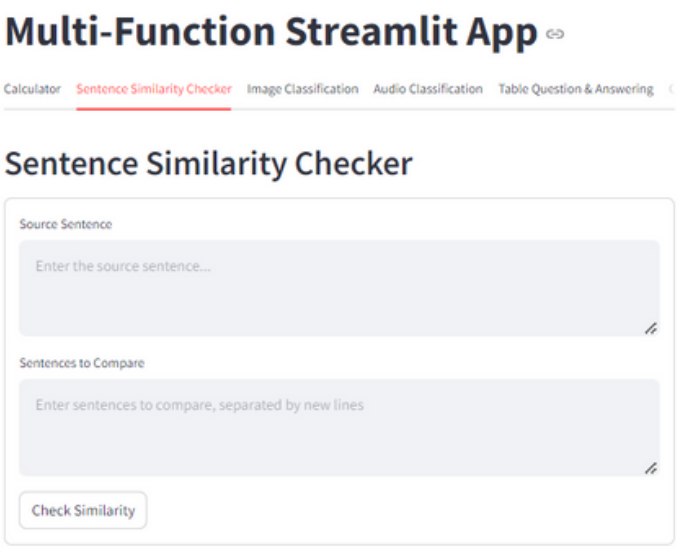
- The similarity function calculates the similarity scores between the source sentence and the list of comparison sentences. It includes:
- Mean Pooling: A method to compute the average of token embeddings while considering the attention mask.
- Get Similarity Function: This function encodes the input sentences, retrieves embeddings, normalizes them, and computes cosine similarity scores.

4 Streamlit Application Structure:

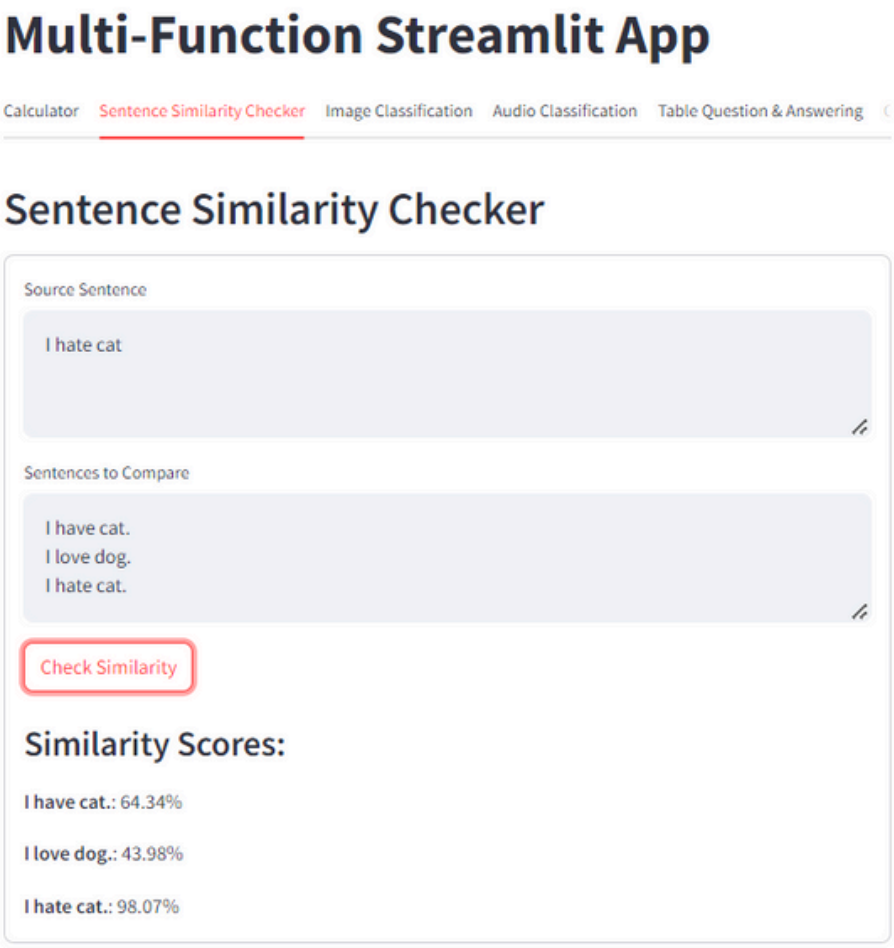
- The main part of the application uses Streamlit to create an interactive interface:
- It displays a header and a loading spinner while the model is being loaded.
- It contains two text areas for the user to input the source sentence and comparison sentences.
- When the user submits the form, it triggers the similarity check.
- If both sentences are provided, it calculates and displays the similarity scores for the comparison sentences.

5 Error Handling:

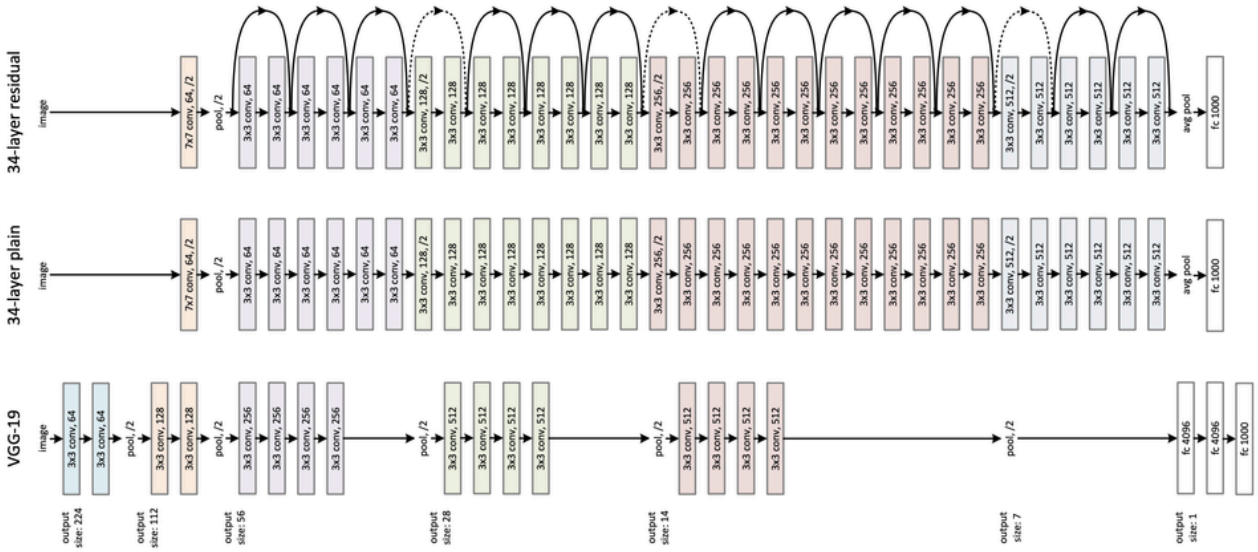
- The application includes basic error handling to display any exceptions that occur during execution.



Output Snippets:



7 IMAGE CLASSIFICATION



ResNet model pre-trained on ImageNet-1k at resolution 224x224. It was introduced in the paper Deep Residual Learning for Image Recognition by He et al.ResNet (Residual Network) is a convolutional neural network that democratized the concepts of residual learning and skip connections. This enables to train much deeper models.

This is ResNet v1.5, which differs from the original model: in the bottleneck blocks which require downsampling, v1 has stride = 2 in the first 1x1 convolution, whereas v1.5 has stride = 2 in the 3x3 convolution. This difference makes ResNet50 v1.5 slightly more accurate (~0.5% top1) than v1, but comes with a small performance drawback (~5% imgs/sec) according to Nvidia.

Source Code:

```python
def image_classifier(image):
    # API details
    API_URL = "https://api-inference.huggingface.co/models/timm/resnet50.a1_in1k"
    headers = {"Authorization": "Bearer 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"} # Replace with your actual token if needed

    # Function to query the model
    def query(image_data):
        response = requests.post(API_URL, headers=headers, data=image_data)
        if response.status_code == 200:
            return [result["label"] for result in response.json()]
        else:
            st.error(f"Error: {response.status_code} - {response.text}")
            return []

    return query(image)
```

```python
# Image Classification tab
with tab3:
  try:
    st.header("Image Classification with Hugging Face Model")
    st.markdown("Upload an image to classify using the ResNet50 model.")

    # Upload image
    uploaded_file = st.file_uploader("Choose an image...", type = ["jpg", "jpeg", "png","webp"])

    if uploaded_file is not None:
        # Open and display the image
        image = Image.open(uploaded_file)
        st.image(image, caption='Uploaded Image', width = 300)

        # Convert image to bytes for the request
        image_bytes = io.BytesIO()
        image.save(image_bytes, format='WEBP')
        image_bytes.seek(0)  # Seek to the start of the BytesIO buffer

        # Get classification
        if st.button("Classify Image"):
            with st.spinner("Classifying..."):
                results = image_classifier(image_bytes.read())
                if results:
                    st.success("Classification Results:")
                    for label in results:
                        st.write(label)
  except Exception as e:
    st.error(f"An error occurred: {e}")
```

Pipeline:

1 Import Required Libraries:

- The code imports necessary libraries, including Streamlit for the user interface, requests for making API calls, and PIL (Python Imaging Library) for image handling.

2 Image Classifier Function:

- The image_classifier function defines the API details and implements a query function that sends the uploaded image to the Hugging Face model for classification. The function handles the API response and returns the classification results.

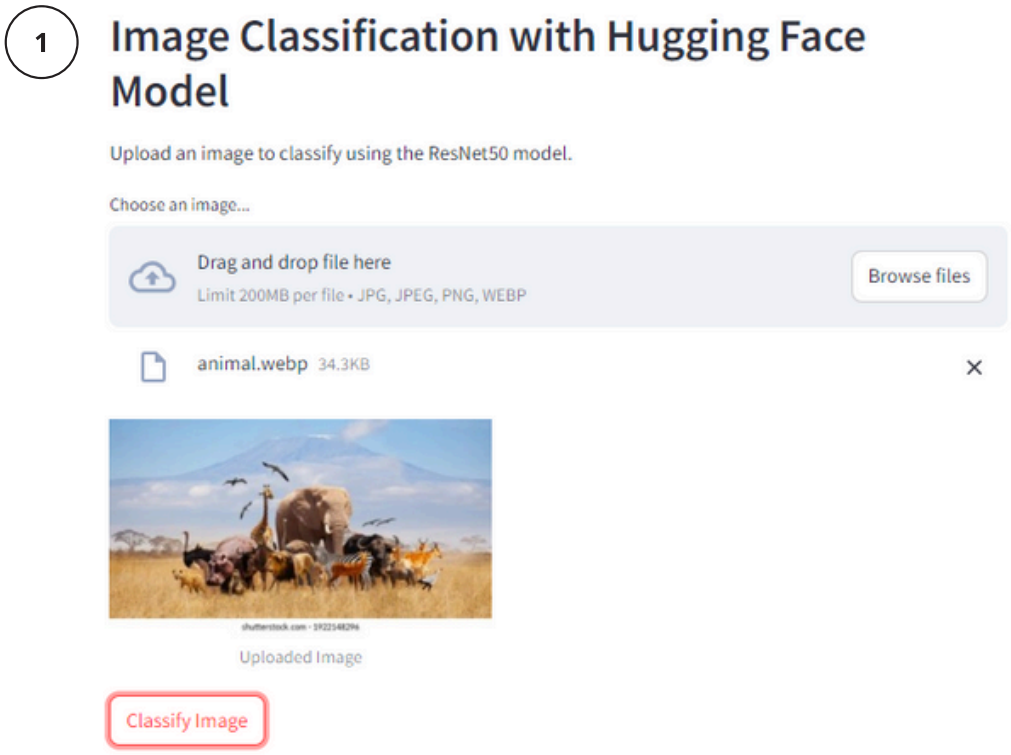
4 **Streamlit Application Structure:**

- The main part of the application creates an interactive interface using Streamlit:
    - It displays a header and a description for the image classification task.
    - A file uploader allows users to upload images in specified formats (JPG, JPEG, PNG, WEBP).
    - If an image is uploaded, it is displayed on the screen.
    - When the user clicks the "Classify Image" button, the application reads the image as bytes and sends it to the image classifier.

5 **Error Handling:**

- The application includes basic error handling to display any exceptions that occur during execution, ensuring users receive feedback in case of issues.

Output Snippets:



2

Calculator | Sentence Similarity Checker | Image Classification | Audio Classification | Table Question & Answering

Image Classification with Hugging Face Model

Upload an image to classify using the ResNet50 model.

Choose an image...

Drag and drop file here
Limit 200MB per file • JPG, JPEG, PNG, WEBP

Browse files

animal.webp 34.3KB

Uploaded Image

Classify Image

Classification Results:

African elephant, Loxodonta africana: an elephant native to Africa having enormous flapping ears and ivory tusks

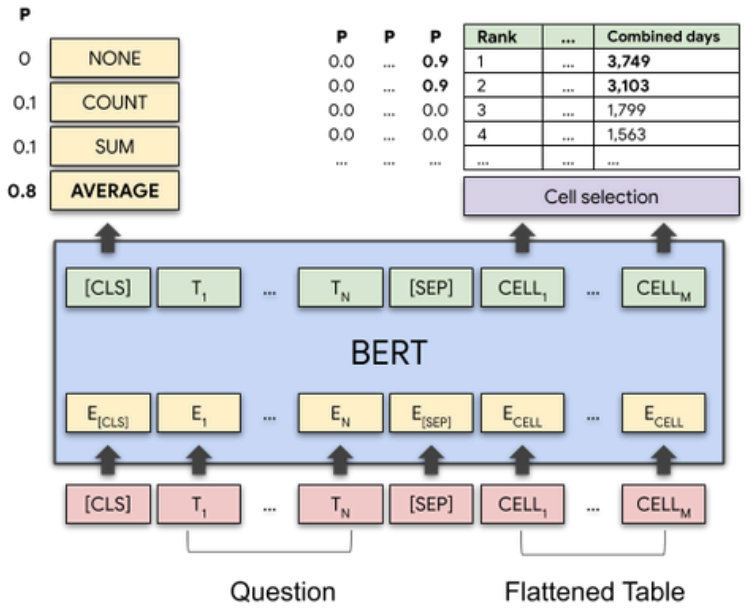
zebra: any of several fleet black-and-white striped African equines

hartebeest: a large African antelope with lyre-shaped horns that curve backward

gazelle: small swift graceful antelope of Africa and Asia having lustrous eyes

hog, pig, grunter, squealer, Sus scrofa: domestic swine

8 TABLE QUESTION & ANSWERING



This model has 2 versions which can be used. The default version corresponds to the tapas_wtq_wikisql_sqa_inter_masklm_large_reset checkpoint of the original Github repository. This model was pre-trained on MLM and an additional step which the authors call intermediate pre-training, and then fine-tuned in a chain on SQA, WikiSQL and finally WTQ. It uses relative position embeddings (i.e. resetting the position index at every cell of the table).

I used BERT-like transformers model pretrained on large corpus of English data from Wikipedia in a self-supervised fashion. This means it was pretrained on the raw tables and associated text only, with no humans labelling them in any way with an automatic process to generate inputs and labels from those texts. More precisely, it was pretrained with two objectives.

Source Code:

```
#Table Q&A:
def table_qa(table, question,qa_tokenizer, qa_model, qa_pipe):

  def preprocess_table(table):
      """
      Preprocesses the table by converting all cells to strings
      and filling any missing values.
      """
      return table.fillna("").astype(str)

  def answer_question(table, question):
      """
      Answers a question based on the provided table.

      Parameters:
          table (pd.DataFrame): Data table in pandas DataFrame format.
          question (str): Question about the table.

      Returns:
          str: Answer to the question.
      """
      # Preprocess the table to ensure compatibility
      table = preprocess_table(table)

      # Use the pipeline to answer the question based on the table
      try:
          answer = pipe({"table": table.to_dict(orient="records"), "query": question})
          return answer["answer"]
      except Exception as e:
          st.error(f"An error occurred: {e}")
          return "An error occurred during processing."

  # Assign the loaded TAPAS model and tokenizer
  pipe = qa_pipe
  tokenizer = qa_tokenizer
  model = qa_model

  return answer_question(table, question)
```

```
#Table Question & Answering:
with tab5:
  try:
    # Streamlit Interface
    st.title("Table Question Answering with TAPAS Model")
    with st.spinner("Loading model..."):
        qa_tokenizer, qa_model, qa_pipe = load_table_qa()

    # Table upload or data entry
    uploaded_file = st.file_uploader("Upload a CSV file containing the table data")
    if uploaded_file:
        table_data = pd.read_csv(uploaded_file)
    else:
        st.warning("Please upload a CSV file.")

    if uploaded_file:
        st.write("### Table Data")
        st.dataframe(table_data)

        # Question input
        question = st.text_input("Enter your question about the table:")

        # Display the answer
        if st.button("Get Answer"):
            if question:
                with st.spinner("Processing..."):
                    answer = table_qa(table_data, question ,qa_tokenizer, qa_model, qa_pipe)

                    st.success(f"Answer: {answer}")
            else:
                st.warning("Please enter a question.")
  except Exception as e:
      st.error(f"An error occurred: {e}")
```

Pipeline:

Define Helper Functions:

load_table_qa():

- Loads the TAPAS question-answering model and tokenizer using the Hugging Face Transformers library.
- Creates a question-answering pipeline specifically for table data.

preprocess_table():

- Ensures all cells in the table are strings and fills any missing values.

answer_question():

- Takes a table and a question as input.
- Processes the table to make it compatible with the model.
- Uses the QA pipeline to generate an answer.
- Returns the answer or an error message if processing fails.

Streamlit App (table_qa() function):

- Sets the title "Table Question Answering with TAPAS Model" in the Streamlit app
- Shows a loading spinner while the TAPAS model, tokenizer, and pipeline are loaded through load_table_qa().
-

Generate Answer:

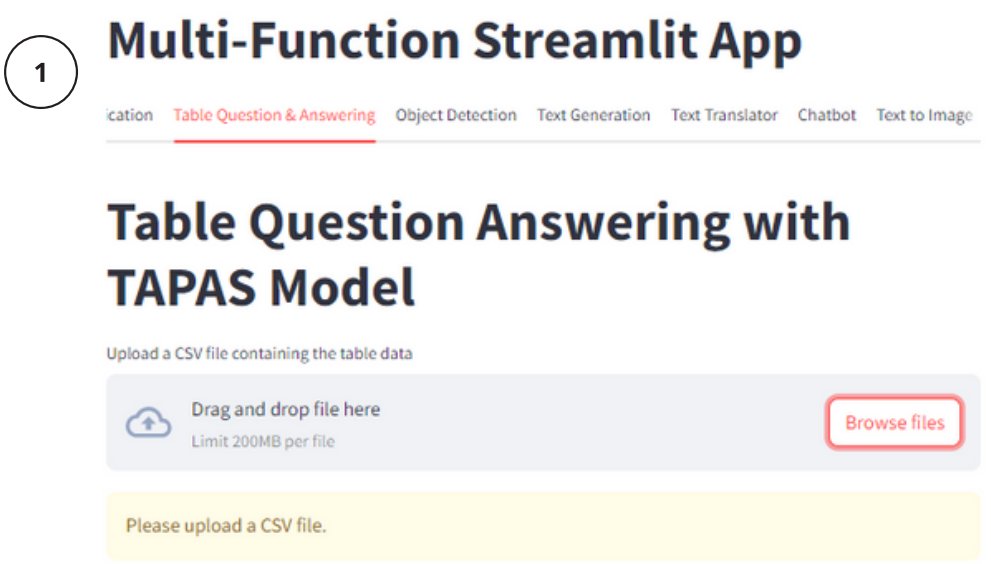
- Displays a "Get Answer" button.
- If a question is entered and the button is pressed:
- Shows a processing spinner.
- Calls answer_question() with the table data, question, and QA pipeline to generate an answer.

Exception Handling:

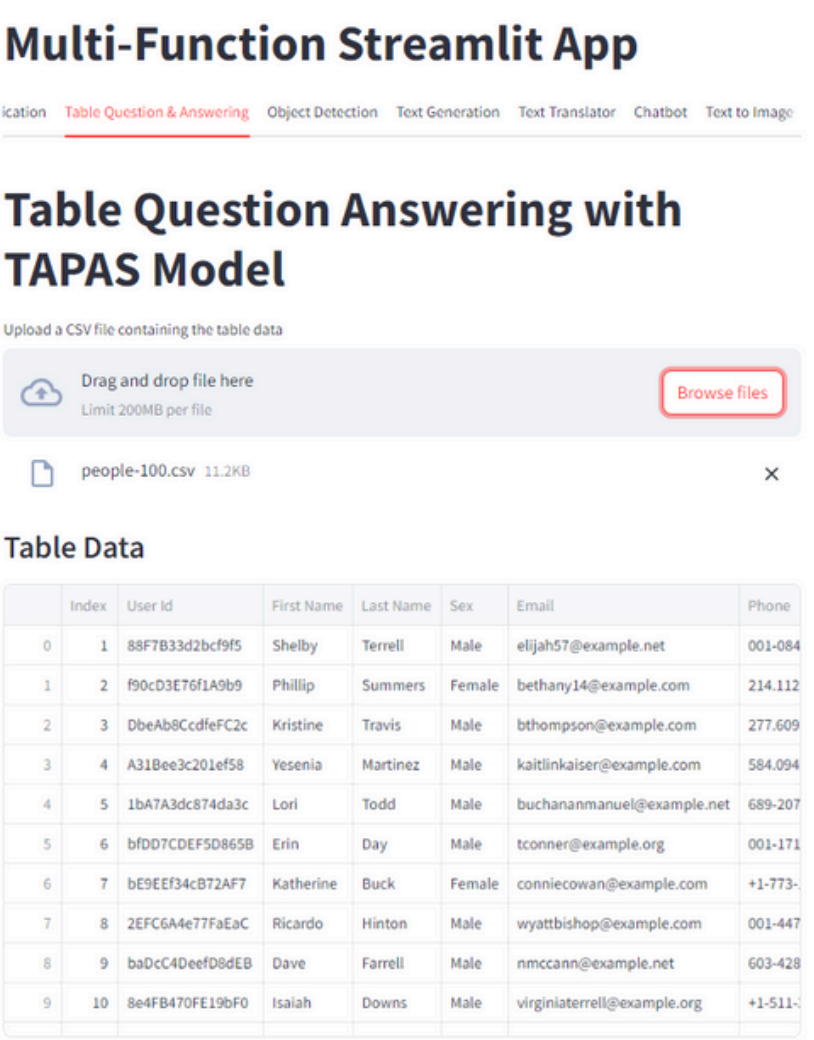
- If any error occurs during processing, displays an error message in the app interface.
- Run the table_qa() Function:
- Calls table_qa() to start the Streamlit app, allowing users to upload a table, ask questions, and receive answers.

Output Snippets:

1



2



	Index	User Id	First Name	Last Name	Sex	Email	Phone
0	1	88F7B33d2bcf9f5	Shelby	Terrell	Male	elijah57@example.net	001-084
1	2	f90cD3E76f1A9b9	Phillip	Summers	Female	bethany14@example.com	214.112
2	3	DbeAb8CcdfeFC2c	Kristine	Travis	Male	bthompson@example.com	277.609
3	4	A31Bee3c201ef58	Yesenia	Martinez	Male	kaitlinkaiser@example.com	584.094
4	5	1bA7A3dc874da3c	Lori	Todd	Male	buchananmanuel@example.net	689-207
5	6	bfDD7CDEF5D865B	Erin	Day	Male	tconner@example.org	001-171
6	7	bE9EEf34cB72AF7	Katherine	Buck	Female	conniecowan@example.com	+1-773-
7	8	2EFC6A4e77FaEaC	Ricardo	Hinton	Male	wyattbishop@example.com	001-447
8	9	baDcC4DeefD8dEB	Dave	Farrell	Male	nmccann@example.net	603-428
9	10	8e4FB470FE19bF0	Isaiah	Downs	Male	virginiaterrell@example.org	+1-511-

Table Data

	Index	User Id	First Name	Last Name	Sex	Email	Phone
0	1	88F7B33d2bcf9f5	Shelby	Terrell	Male	elijah57@example.net	001-084
1	2	f90cD3E76f1A9b9	Phillip	Summers	Female	bethany14@example.com	214.112
2	3	DbeAb8CcdfeFC2c	Kristine	Travis	Male	bthompson@example.com	277.609
3	4	A31Bee3c201ef58	Yesenia	Martinez	Male	kaitlinkaiser@example.com	584.094
4	5	1bA7A3dc874da3c	Lori	Todd	Male	buchananmanuel@example.net	689-207
5	6	bfDD7CDEF5D865B	Erin	Day	Male	tconner@example.org	001-171
6	7	bE9EEf34cB72AF7	Katherine	Buck	Female	conniecowan@example.com	+1-773-
7	8	2EFC6A4e77FaEaC	Ricardo	Hinton	Male	wyattbishop@example.com	001-447
8	9	baDcC4DeefD8dEB	Dave	Farrell	Male	nmccann@example.net	603-428
9	10	8e4FB470FE19bF0	Isaiah	Downs	Male	virginiaterrell@example.org	+1-511-

Enter your question about the table:

what is the email of the person whose User Id is 1bA7A3dc874da3c

Get Answer

Answer: buchananmanuel@example.net

CONCLUSIONS

The conclusion of this project highlights its success in creating a comprehensive, accessible platform that consolidates multiple AI, machine learning, and deep learning models into one user-friendly application. By integrating tools like Hugging Face, Google Console API, and Gemini API into a Streamlit app, the project significantly contributes to making advanced AI functionalities readily accessible to users of varying skill levels. Throughout this project, practical knowledge was gained in deploying and managing pre-trained models, API integration, and UI/UX considerations for an efficient and intuitive user experience. The hands-on experience with these technologies deepened the understanding of how to develop cohesive AI applications, bridging technical functions with real-world usability. Additionally, this project opens doors for further exploration, such as expanding model functionalities, enhancing multilingual support, or exploring custom model fine-tuning. In summary, the project not only serves as an educational tool but also establishes a foundation for future advancements in democratizing AI technology for broader audiences.

REFERENCE

The references section provides a comprehensive list of all sources consulted throughout the project, encompassing technical documentation, libraries, and APIs that were instrumental in the development process. It includes key resources for understanding and utilizing Streamlit, which facilitated the creation of the app's user interface, as well as extensive documentation from Hugging Face that provided guidance on implementing and integrating pre-trained models. Additionally, references include Google Console API and Gemini API documentation, which offered the essential information needed for effective API setup, secure handling of credentials, and usage best practices. By consolidating these references, this section serves as both a record of the project's research foundation and a resource for future developers seeking to build similar AI applications. These sources collectively enabled the integration of cutting-edge AI and machine learning functionalities, helping ensure a seamless, functional, and educational application for users.