PROJECT REPORT

On

Vision Text Bridge – A Multi modal Database Interaction Framework

Team No: 09
Mutheeswaran R | Boya Pavani

Topic : Visual Search Using VLM's

Project Title: Vision Text Bridge – A Multi modal Database Interaction Framework

Problem Statement :

As data grows increasingly complex, users are challenged to interact with and analyze multimedia datasets—such as a combination of text, images, and tabular information—effectively. While traditional database APIs are adept at handling structured data, they lack the capability to process visual inputs or combine insights from textual and visual contexts. This gap poses significant barriers in fields like data visualization, automated reporting, and multi modal analytics.

Project Overview:

Vision Text Bridge is a next-generation Visual Language Model (VLM)-powered platform designed to act as a bridge between multi modal data and databases. It incorporates components for computer vision, natural language processing (NLP), and structured data querying. Users can query the system using text or visual prompts and retrieve results augmented by multi modal reasoning.

Key Capabilities:

1. Visual Data Processing:

- Analyze and extract content from images (e.g., charts, tables, or scanned documents) using vision-based models like ViLBERT or Visual BERT.

- Enable querying of image datasets for object detection, content extraction, or metadata-based searches.

2. Multi modal Integration:

- Combine insights from visual and textual data to perform complex analytics and reasoning, leveraging the power of multi modal VLMs.

3. Structured Data Interaction:

- Provide traditional database operations for structured datasets, with enhanced functionality to handle multi modal contexts.

Libraries used:

Pandas (import pandas as pd):

Category: Data manipulation and analysis.

Relation to VLM: While not directly related to VLMs, Pandas is often used for data handling and preprocessing, which is essential when working with datasets for training or evaluating VLMs.

Pillow (PIL) (from PIL import Image):

Category: Image processing.

Relation to VLM: Image processing is crucial in VLMs, as they often require image inputs. Pillow is used for loading, manipulating, and saving images.

BytesIO (from io import BytesIO):

Category: In-memory binary streams.

Relation to VLM: Used for handling image data in memory, which can be useful when preparing data for VLMs.

Math (import math):

Category: Mathematical functions.

Relation to VLM: While not specific to VLMs, mathematical operations are often necessary in various computations, including image resizing and normalization.

Base64 (import base64):

Category: Encoding and decoding data.

Relation to VLM: Base64 encoding is often used to convert binary data (like images) into a text format, which can be useful for data transmission or storage.

Transformers (from transformers import AutoImageProcessor, ResNetForImageClassification):

Category: Natural Language Processing (NLP) and Vision.

Relation to VLM: The transformers library by Hugging Face includes models for both NLP and vision tasks. The AutoImageProcessor and ResNetForImageClassification can be part of a pipeline that processes images for tasks that may involve language understanding, making them relevant to VLMs.

Dotenv (from dotenv import load_dotenv):

Category: Environment variable management.

Relation to VLM: Not directly related to VLMs, but useful for managing configuration settings, such as API keys or model paths.

Qdrant Client (from qdrant_client import QdrantClient):

Category: Vector database management.

Relation to VLM: Qdrant is used for managing and querying vector embeddings, which are often generated by VLMs. It can be used to store and retrieve embeddings for images and text, making it relevant in the context of VLMs

Installation of Streamlit App:

Step 1: Install Streamlit

1. Install Streamlit using pip:

bash

pip install streamlit

Step 2: Create a New Python File

1. Create a new Python file, e.g., `app.py`.

Step 3: Import Streamlit and Other Libraries

1. Import Streamlit and other required libraries:

python

```
import streamlit as st
import pandas as pd
import numpy as np
```

Step 4: Create a Streamlit App

1. Create a Streamlit app:

st.title("Vision Text Bridge")

st.write("A Multi-modal Database Interaction Framework")

Step 5: Add Interactive Elements

1. Add interactive elements, such as text inputs, sliders, and buttons:

python

```
text_input = st.text_input("Enter your query")
slider_value = st.slider("Select a value", 0, 100)
button_clicked = st.button("Submit")
```

Step 6: Integrate with Vision Text Bridge

1. Integrate the Streamlit app with the Vision Text Bridge project:

```
# Call the Vision Text Bridge functions here
```

Step 7: Run the Streamlit App

1. Run the Streamlit app:

bash

```
streamlit run app.py
```

CODE :

```
import os
from pandas import DataFrame
from PIL import Image
from io import BytesIO
import math
import base64
from transformers import AutoImageProcessor, ResNetForImageClassification
from dotenv import load_dotenv  # type: ignore
from qdrant_client import QdrantClient
from qdrant_client import models


# Load environment variables
load_dotenv()


# Set the absolute base directory
```

```
base_directory = 'C:/Users/muthe/OneDrive/Desktop/int pro/21-streamlit-qdrant-app/21-streamlit-base-project/ipuyb/afhq/3/ver'

# Check if the directory exists
if not os.path.exists(base_directory):
    print(f"Directory does not exist: {base_directory}")
else:
    # Get image URLs from the specified directory
    all_image_urls = os.listdir(base_directory)

    # Sample the first 500 image URLs
    sample_image_urls = all_image_urls[:500]

    # Create a list of absolute paths for the sampled images
    sample_image_urls = list(map(lambda item: os.path.join(base_directory, item), sample_image_urls))

    # Print the sample image URLs
    print(sample_image_urls)

    # Create a DataFrame for payloads
    payloads = DataFrame({"image_url": sample_image_urls})
    payloads["type"] = "3/ver"

    # Function to resize images
    target_width = 256

    def resize_image(image_url):
        pil_image = Image.open(image_url)
        image_aspect_ratio = pil_image.width / pil_image.height
        resized_pil_image = pil_image.resize(
            [target_width, math.floor(target_width * image_aspect_ratio)]
        )
        return resized_pil_image
```

```
    # Function to convert images to base64
    def convert_image_to_base64(pil_image):
        image_data = BytesIO()
        pil_image.save(image_data, format="JPEG")
        base64_string = base64.b64encode(image_data.getvalue()).decode("utf-8")
        return base64_string

    # Resize images and convert to base64
    resized_images = list(map(resize_image, sample_image_urls))
    base64_strings = list(map(convert_image_to_base64, resized_images))
    payloads["base64"] = base64_strings

    # Load the model and processor
    processor = AutoImageProcessor.from_pretrained("microsoft/resnet-50")
    model = ResNetForImageClassification.from_pretrained("microsoft/resnet-50")

    # Prepare inputs for the model
    inputs = processor(
        resized_images,  # Use resized_images instead of images
        return_tensors="pt",
    )

    # Get embeddings from the model
    outputs = model(**inputs)
    embeddings = outputs.logits.detach().numpy()  # Convert to numpy array
    embedding_length = len(embeddings[0])

    # Initialize Qdrant client
    qclient = QdrantClient(
        url=os.getenv('https://fd8db3a8-183a-4e98-af13-cecf72735bf7.europe-west3-
0.gcp.cloud.qdrant.io'),  # Use environment variable for URL

api_key=os.getenv('eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhY2Nlc3MiOiJtIn0.4MNgRBlp6E0N3vURX
A-GztpGiBDNMR56wS4mb90aMbc')  # Use environment variable for API key
```

```
)

# Test the connection to Qdrant
try:
    qclient.health()  # Check the health of the Qdrant server
    print("Connection to Qdrant successful!")
except Exception as e:
    print(f"Connection to Qdrant failed: {e}")

# Create a collection in Qdrant
collection_name = "animal_images"
collection = qclient.create_collection(
    collection_name=collection_name,
    vectors_config=models.VectorParams(
        size=embedding_length,
        distance=models.Distance.COSINE
    )
)

# Convert payloads to a list of dictionaries
payload_dicts = payloads.to_dict(orient="records")

# Prepare records for upload
records = [
    models.Record(
        id=idx,
        payload=payload_dicts[idx],
        vector=embeddings[idx].tolist()  # Convert to list for Qdrant
    )
    for idx in range(len(payload_dicts))
]

# Upload records to Qdrant
qclient.upload_records(
```

```
    collection_name=collection_name,
    records=records
)
```

Core Modules and Code Description

1. Visual Parser Module:

- Purpose: Process and extract data from images, such as detecting tables, charts, or annotated visuals.
- Key Libraries: OpenCV, PyTorch, and Vision Transformers.
- Sample Code:

```
python
from transformers import ViTForImageClassification, ViTFeatureExtractor
from PIL import Image

Load pre-trained Vision Transformer model
feature_extractor =ViTFeatureExtractor.from_pretrained('google/vit-base-patch16-224')
model = ViTForImageClassification.from_pretrained('google/vit-base-patch16-224')
Process an image and predict content
image = Image.open("chart.png")
inputs = feature_extractor(images=image, return_tensors="pt")
outputs = model(**inputs)
predictions = outputs.logits.argmax(-1)
print(f"Predicted label: {predictions}")
```

2. Text Query Processor:

- Purpose: Interpret natural language queries and translate them into operations for multi modal reasoning or database interaction.
- Key Libraries: Hugging Face Transformers, spaCy.
- Sample Code:

```
python
from transformers import pipeline

Load a pre-trained NLP model
```

question_answering = pipeline('question-answering', model='bert-large-uncased-whole-word-masking-finetuned-squad')

query = "Which table contains sales data?"
context = "The sales data is stored in the 'Sales_Table' database table."
result = question_answering(question=query, context=context)
print(f"Answer: {result['answer']}")

3. Multi modal Fusion Module:

- Purpose: Combine insights from visual and textual analysis for comprehensive reasoning.
- Key Libraries: MultiModal Transformers, ViLBERT.
- Sample Code:

python

Example pseudocode for combining visual and text embeddings
text_embedding = nlp_model.encode("Retrieve data from the bar chart")
visual_embedding = vision_model.encode(image)
combined_representation = fusion_model.combine(text_embedding, visual_embedding)

result = combined_representation.analyze()
print(f"Combined insights: {result}")

4. Database Interaction Module:

- Purpose: Perform structured queries on relational databases based on inputs from text and visual modules.
- Key Libraries: SQLAlchemy, Pandas.
- Sample Code:

python
import sqlalchemy

Connect to a database
engine = sqlalchemy.create_engine("sqlite:///database.db")
with engine.connect() as conn:
    result = conn.execute("SELECT * FROM Sales_Table WHERE Region='North'")
    for row in result:

```
print(row)
```

Qdrant can be effectively used in Visual Language Models (VLMs) as a vector database for storing and retrieving embeddings generated by multimodal models. It plays a crucial role in enhancing the capabilities of VLMs, especially in tasks like retrieval-augmented generation (RAG), multimodal search, and contextual reasoning.

How Qdrant Fits into VLMs:

1. Vector Storage:

- Qdrant stores high-dimensional embeddings generated by VLMs, such as those combining visual and textual features.
- These embeddings can represent images, text, or multimodal data, enabling efficient similarity searches.

2. Contextual Retrieval:

- Qdrant can retrieve relevant embeddings based on input queries, providing context for VLMs to generate accurate and informed responses.
- For example, in a multimodal search, Qdrant can retrieve embeddings of images and text that match a user's query.

3. Integration with RAG Pipelines:

- Qdrant is often used in RAG pipelines, where it acts as a knowledge base to enrich prompts sent to VLMs.
- This integration ensures that VLMs have access to relevant data, improving their reasoning and output quality.

4. Multi modal Applications:

- Qdrant can be used in applications like visual document retrieval, multimodal recommendation systems, and interactive AI systems that combine vision and language.

Expected Outcomes :

- A VLM-enabled system capable of intelligent querying and reasoning across multimodal datasets.
- An extensible framework for combining visual, textual, and structured data insights.
- Empowered workflows for domains like automated reporting, visual analytics, and database exploration.