پروژه کارشناسی:
Multimodal RAG

نام دانشجو: حسین آریان‌مهر
شماره دانشجویی: ۴۰۱۱۲۶۲۱۶۷
استاد: دکتر هادی صدوقی یزدی

۱ فهرست

فهرست مطالب

۲ نحوه *setup* پروژه و اجرای کد

۱.۲ ۱ – نصب داکر

فایل نصبی داکر را از وب‌سایت رسمی آن دانلود و نصب کنید.
لینک سایت:

`https://www.docker.com/products/docker-desktop/`

پس از نصب، سیستم را ری‌استارت کنید.

۲.۲ ۲ – سورس‌کد پروژه

لینک پروژه در گیت:

`https://github.com/HosseinArianmehr2004/Multimodal-RAG-System`

برای دسترسی به سورس‌کد پروژه می‌توانید از هر یک از دو راه زیر استفاده کنید:
A – استفاده از :fork به آدرس داده شده در بالا بروید و در قسمت بالا سمت چپ صفحه روی فلش کنار عنوان *fork* کلیک کنید و گزینه‌ی *create a new fork* را انتخاب کرده و مراحل را پیش ببرید.
B – استفاده از :clone با استفاده از دستور زیر، سورس‌کد پروژه را در سیستم لوکال خود کلون کنید:

```
git clone project_link path
```

۳.۲ ۳ – نصب محیط مجازی و کتابخانه‌های موردنیاز

در پوشه‌ی پروژه (*PROJECT_ROOT*) با دستورات زیر یک محیط مجازی ایجاد کنید. یک *CMD* باز کنید و به مسیر *root* پروژه بروید. سپس دستورات زیر را اجرا کنید:

```
python -m venv environment_name
environment_name\Scripts\activate
```

پس از اکتیو شدن محیط مجازی با دستور زیر تمام کتابخانه‌های موردنیاز را نصب کنید:

```
pip install -r requirements.txt
```

به دلیل تعداد زیاد کتابخانه‌ها و همچنین وجود کتابخانه‌های پرحجمی مانند *pytorch* در بین آن‌ها، زمان نصب ممکن است کمی طول بکشد.

۴.۲ ۴ – ساخت دیتابیس *Weaviate*

در ابتدا نرم‌افزار *Docker* را در ویندوز اجرا کنید. سپس یک *CMD* باز کرده و به مسیر *root* پروژه بروید. برای فعال‌سازی سرویس *Weaviate* در داکر، دستور زیر را اجرا کنید:

```
docker-compose up -d
```

این دستور فایل *docker-compose.yml* را اجرا کرده و سرویس *Weaviate* را روی داکر راه‌اندازی می‌کند.

۱.۴.۲ توضیح فایل *docker-compose.yml*

فایل *docker-compose.yml* وظیفه دارد نحوه‌ی استقرار و اجرای سرویس‌های مختلف پروژه را در محیط *Docker* مشخص کند. در این پروژه، تنها یک سرویس با نام *Weaviate* تعریف شده است. محتوای این فایل به شرح زیر است:

```
version: '3.4'
services:
  weaviate:
    image: semitechnologies/weaviate:latest
    restart: always
    ports:
      - "8080:8080"
      - "50051:50051"
    environment:
      QUERY_DEFAULTS_LIMIT: 25
      AUTHENTICATION_ANONYMOUS_ACCESS_ENABLED: 'true'
      PERSISTENCE_DATA_PATH: "./data"
      DISABLE_MODULES: 'all'
```

توضیحات بخش‌های مختلف:

- **version**: نسخه‌ی نگارش فایل *docker-compose* را مشخص می‌کند. در نسخه‌های جدید *Docker* این گزینه اختیاری است و می‌توان آن را حذف کرد.
- **services**: لیست سرویس‌هایی است که باید توسط *Docker* اجرا شوند. در اینجا تنها سرویس *weaviate* تعریف شده است.
- **weaviate**: نام سرویس مربوط به پایگاه‌داده‌ی برداری (*Weaviate*) است.
- **image**: مشخص می‌کند که سرویس از تصویر (*Image*) رسمی *Weaviate* با آخرین نسخه (*latest*) استفاده می‌کند. این تصویر از مخزن *Docker Hub* دریافت می‌شود.
- **restart**: با مقدار *always* تعیین می‌شود که در صورت توقف یا بروز خطا، کانتینر به‌صورت خودکار مجدداً راه‌اندازی شود.
- **ports**: نگاشت پورت‌های داخلی کانتینر به پورت‌های میزبان را تعیین می‌کند:
  - پورت *8080* برای دسترسی به رابط *HTTP* و *API* اصلی *Weaviate*.
  - پورت *50051* برای ارتباطات *gRPC*.
- **environment**: شامل متغیرهای محیطی است که پارامترهای اجرایی *Weaviate* را پیکربندی می‌کنند:
  - *QUERY_DEFAULTS_LIMIT: 25* — تعداد نتایج پیش‌فرض در هر Query را ۲۵ تعیین می‌کند.
  - *AUTHENTICATION_ANONYMOUS_ACCESS_ENABLED: 'true'* — امکان دسترسی ناشناس به *Weaviate* را فعال می‌سازد (بدون نیاز به احراز هویت).
  - *PERSISTENCE_DATA_PATH: "./data"* — مسیر ذخیره‌سازی داده‌های پایدار را مشخص می‌کند تا داده‌ها حتی پس از توقف کانتینر حذف نشوند.
  - *DISABLE_MODULES: 'all'* — تمام ماژول‌های اختیاری *Weaviate* (از جمله مدل‌های بردارساز) را غیرفعال می‌کند و تنها هسته‌ی اصلی پایگاه‌داده اجرا می‌شود.

نتیجه‌ی اجرا: با اجرای دستور زیر:

```
docker-compose up -d
```

سرویسی تحت عنوان *Weaviate* در محیط *Docker* اجرا می‌شود که از طریق آدرس‌های زیر در دسترس است:

- *HTTP API: http://localhost:8080*
- *gRPC API: localhost:50051*

این تنظیمات باعث می‌شود *Weaviate* به‌صورت پایدار و مستقل در بستر *Docker* اجرا گردد، داده‌ها در مسیر محلی ذخیره شوند و امکان ارتباط از طریق پورت‌های مشخص‌شده فراهم گردد.

۵.۲ ۵ – گرفتن API

برای استفاده از مدل‌های *LLM* نیاز به *API* آن‌ها دارید. در این پروژه، *API*های مدنظر از *OpenRouter* گرفته شده‌اند.
برای گرفتن *API* به طریق زیر عمل کنید:

۱. به وب‌سایت رسمی *OpenRouter* بروید: `https://openrouter.ai/`

۲. در صورتی که حساب کاربری ندارید، ابتدا یک حساب کاربری بسازید و لاگین کنید.

۳. روی علامت منو در بالا سمت راست صفحه کلیک کرده و گزینه‌ی *keys* را انتخاب کنید.

۴. در صفحه‌ی مورد نظر، روی دکمه‌ی *Create API Key* کلیک کنید.

۵. یک نام برای کلید خود انتخاب کرده و گزینه‌ی *Create* را بزنید.

۶. سپس یک *pop-up* به شما نمایش داده می‌شود. از همین قسمت کلید خود را کپی و در یک فایل ذخیره کنید.

توجه کنید که پس از بستن *pop-up*، نمی‌توانید به محتوای کلید دست یابید. سپس باید کلید خود را در متغیرهای محیطی پروژه ذخیره کنید تا در کد از آن استفاده شود.
برای این کار، به پوشه‌ی پروژه رفته و در آنجا فایلی با نام *env*. بسازید و خط زیر را در آن کپی کنید. سپس کلید خود را در قسمت *your_key* قرار دهید:

```
LLM_API_KEY=your_key
```

در پایان، فایل را ذخیره کنید.

۳ ایجاد Collection در پایگاه‌داده Weaviate

برای ذخیره‌سازی داده‌ها در پایگاه‌داده باید یک *collection* ایجاد کنید. برای این کار ابتدا یک cmd باز کرده و به مسیر root پوژه بروید، سپس اسکریپت *create_database_collection.py* را به‌صورت زیر اجرا نمایید:

```
python import_data_into_database/create_database_collection.py
```

۱.۳ توضیح کد پایتون اتصال و پیکربندی Weaviate

کد زیر به‌منظور اتصال به پایگاه‌داده‌ی برداری *Weaviate*، ایجاد یک مجموعه (*Collection*) و تعریف ساختار داده‌ها در آن نوشته شده است. این کد از کتابخانه‌ی رسمی *weaviate-client* در پایتون استفاده می‌کند.

```
import weaviate
import weaviate.classes as wvc
```

دو ماژول اصلی کتابخانه *Weaviate* فراخوانی می‌شوند:

- *weaviate*: برای مدیریت اتصال و انجام عملیات اصلی.
- *weaviate.classes*: برای تعریف تنظیمات و ساختار مجموعه‌ها (*Collections*) و ویژگی‌ها (*Properties*).

۲.۳ اتصال به Weaviate

```
try:
    client = weaviate.connect_to_local()
    print("Successfully connected to Weaviate!")
except Exception as e:
    print(f"Failed to connect to Weaviate: {e}")
    exit()
```

در این بخش، با استفاده از تابع *connect_to_local()* یک اتصال به سرویس محلی *Weaviate* برقرار می‌شود (یعنی همان سرویسی که با *Docker* روی *localhost:8080* اجرا شده است).

۳.۳ تعریف نام مجموعه (Collection)

```
collection_name = "Multimodal_Collection"
```

در این خط، نام مجموعه‌ای که قرار است در پایگاه داده ایجاد شود مشخص می‌گردد. در *Weaviate*، مجموعه‌ها مشابه «جداول» در پایگاه‌داده‌های رابطه‌ای هستند.

۴.۳ حذف مجموعه‌ی قبلی (در صورت وجود)

```
if client.collections.exists(collection_name):
    print(f"Collection '{collection_name}' already exists. Deleting it.")
    client.collections.delete(collection_name)
```

در این بخش ابتدا بررسی می‌شود که آیا مجموعه‌ای با همین نام از قبل وجود دارد یا خیر. اگر وجود داشته باشد، برای جلوگیری از تکرار و تعارض، ابتدا حذف می‌شود.

۵.۳ ایجاد مجموعه‌ی جدید

```
print(f"Creating collection '{collection_name}'...")
my_collection = client.collections.create(
    name=collection_name,
    properties=[
        wvc.config.Property(name="modality", data_type=wvc.config.DataType.TEXT),
        wvc.config.Property(name="content", data_type=wvc.config.DataType.TEXT),
        wvc.config.Property(name="contentId", data_type=wvc.config.DataType.TEXT),
        wvc.config.Property(name="filePath", data_type=wvc.config.DataType.TEXT),
    ]
)
print(f"Collection '{collection_name}' created successfully.")
```

در این مرحله، مجموعه‌ای جدید با نام *Multimodal_Collection* ایجاد می‌شود. پارامتر *properties* مشخص‌کننده‌ی فیلدهای درون این مجموعه است. هر ویژگی با استفاده از کلاس *wvc.config.Property* تعریف می‌شود و نوع داده‌ی آن نیز با *wvc.config.DataType* تعیین می‌گردد.

نام ویژگی	نوع داده	توضیح
modality	TEXT	نوع داده‌ی ورودی (مثلاً متن، تصویر یا صوت)
content	TEXT	محتوای اصلی داده
contentId	TEXT	شناسه‌ی منحصربه‌فرد هر داده یا سند
filePath	TEXT	مسیر فایل اصلی در سیستم محلی یا فضای ذخیره‌سازی

به‌این‌ترتیب، ساختار مجموعه به‌گونه‌ای طراحی شده که از داده‌های چندرسانه‌ای (متن، تصویر و صوت) پشتیبانی کند. در این مجموعه، هر داده شامل فیلدهای بالا است:

داده‌های متنی

```
modality = "text"
content = Original text
contentId = a unique id
filePath = ""
```

داده‌های تصویری

```
modality = "image"
content = ""
contentId = a unique id
filePath = "/content/image_dataset/{file_name}"
```

داده‌های صوتی

```
modality = "audio"
content = Audio transcription
contentId = a unique id
filePath = "/content/audio_dataset/{file_name}"
```

۶.۳ بستن اتصال

```
client.close()
```

پس از اتمام عملیات، اتصال به پایگاه داده بسته می‌شود تا منابع آزاد شوند.

۴ دیتاست

دیتاست‌ها در فضای *Google Drive* ذخیره شده‌اند و از طریق لینک‌های زیر قابل دانلود هستند:

- دیتاست تصاویر:
https://drive.google.com/file/d/1a68xCnTnnRPrBSM7bHSRzHwLBpOoLWuL/view?usp=sharing
- دیتاست فایل‌های صوتی:
https://drive.google.com/file/d/1BS72l_5Z6wvGiq9Sn8zuV4-IfwxLRvR5/view?usp=sharing
- دیتاست متن:
https://drive.google.com/file/d/1RMjyguno7iLfUtlW9gumdwrgbB9ht252/view?usp=sharing

در پوشه‌ی پروژه، یک پوشه‌ی جدید به نام *content* ایجاد کنید. در این پوشه، سه زیرپوشه با نام‌های زیر بسازید و داده‌های مربوطه را در هرکدام قرار دهید:

- *audio_dataset*: شامل تمام فایل‌های صوتی
- *image_dataset*: شامل تمام تصاویر

- *text_dataset*: شامل فایل *CSV* داده‌های متنی

در صورتی‌که قصد دارید از دیتاست‌ها یا داده‌های خود استفاده کنید، روال به همان شکل بالا است. کافی است تمام تصاویر خود را در پوشه‌ی *image_dataset* و تمام فایل‌های صوتی را در پوشه‌ی *audio_dataset* قرار دهید.
برای داده‌های متنی، باید یک فایل *CSV* با ساختار زیر ایجاد نمایید: متن‌های طولانی، فایل‌های *PDF* و سایر متون را به رشته‌های (*string*) جداگانه تقسیم کنید. سپس فایلی با فرمت *CSV* ایجاد کرده و در آن ستونی به نام *text* قرار دهید. تمام داده‌های متنی (رشته‌ها) را در این ستون وارد کنید.

```
text
data 1
data 2
data 3
...
```

داده‌های متنی و صوتی شما می‌توانند به زبان‌های فارسی یا انگلیسی باشند.

۵ مدل *open_clip*

برای استفاده از مدل *open_clip* باید وزن‌های آن را دانلود کنید.
وزن‌ها را می‌توانید از لینک زیر در گوگل‌درایو دریافت کنید:

https://drive.google.com/file/d/1czuHrdWylmzYHA1Rh5NQJsnFDB9BSp7Y/view?usp=sharing

پس از دانلود، فایل را در مسیر زیر قرار دهید:

PROJECT_ROOT/open_clip_weights/ViT-B-32-openai/

۶ وارد کردن داده‌ها به دیتابیس

۱.۶ وارد کردن داده‌های تصویری به دیتابیس

پس از قرار دادن تمام تصاویر در پوشه مربوطه، یک *CMD* باز کنید و به مسیر پروژه بروید. سپس با دستور زیر اسکریپت پایتون با نام `images.py` را اجرا کنید:

```
python import_data_into_database/import_local_data/images.py
```

این اسکریپت تمام داده های داخل پوشه *PROJECT_ROOT/content/image_dataset* را بارگذاری میکند، برای هر کدام یک embedding تولید میکند و سپس آن embedding را به همراه مسیر فایل و چند اطلاعات دیگر در دیتابیس ذخیره میکند

توضیح کد

این اسکریپت در ابتدا با استفاده از کد زیر به دیتابیس و collection ای که از قبل ساخته‌ایم متصل می‌شود:

```
try:
    weaviate_client = weaviate.connect_to_local()
    print("  Connected to Weaviate")
    collection = weaviate_client.collections.get("Multimodal_Collection")
except Exception as e:
    print(f"  Weaviate connection failed: {e}")
    weaviate_client = None
```

در این بخش، با استفاده از تابع `connect_to_local()` یک اتصال به سرویس محلی Weaviate برقرار می‌شود (همان سرویسی که با Docker روی *localhost:8080* اجرا شده است).

تنظیمات مدل CLIP

کد زیر تنظیمات مدل CLIP را انجام می‌دهد؛ این مدل برای تولید *embedding* داده‌ها به کار می‌رود:

```
MODEL_NAME = "ViT-B-32"
PRETRAINED_LOCAL_PATH = (
    "./open_clip_weights/ViT-B-32-openai/open_clip_model.safetensors"
)
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"

clip_model, _, preprocess = open_clip.create_model_and_transforms(
    MODEL_NAME, pretrained=PRETRAINED_LOCAL_PATH
)
```

```
tokenizer = open_clip.get_tokenizer(MODEL_NAME)
clip_model.to(DEVICE).eval()
```

توضیح پارامترها:

- **MODEL_NAME** = *"ViT-B-32"*: نام معماری مدل CLIP (*Vision Transformer Base with patch size* = *32×32*).
- **PRETRAINED_LOCAL_PATH**: مسیر فایل وزن‌های از پیش‌آموزش داده‌شده‌ی مدل.
- **DEVICE**: بررسی وجود GPU (*CUDA*)، در صورت نبود GPU از CPU استفاده می‌شود.
- **preprocess clip_model**: بارگذاری مدل و تابع پیش‌پردازش تصاویر.
- **tokenizer**: توکنایزر مخصوص همان مدل.
- **clip_model.to(DEVICE).eval()**: انتقال مدل به GPU یا CPU و قرار دادن در حالت ارزیابی.

بارگذاری و پردازش تصاویر

```
if __name__ == "__main__":
    image_folder = "./content/image_dataset"
    process_images(image_folder)
    weaviate_client.close()
```

توضیح کد:

- مسیر پوشه حاوی تصاویر مشخص شده است.
- تابع `process_images` برای تمام تصاویر داخل پوشه مربوطه *embedding* تولید می‌کند و آنها را در دیتابیس ذخیره می‌کند.

تابع `process_images`

```
def process_images(image_folder: str, max_workers: int = 8):
    """Process and store image embeddings in parallel."""
    images = [
        file_path
        for file_path in os.listdir(image_folder)
        if file_path.lower().endswith((".jpg", ".jpeg", ".png"))
    ]
    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        futures = [
            executor.submit(
                store_image_item, os.path.splitext(os.path.basename(img_name))[0],
    img_name
            )
            for img_name in images
        ]
    for _ in tqdm(as_completed(futures), total=len(futures), desc="Processing images"):
        pass
```

توضیح تابع

- **ورودی‌ها:**
  - `image_folder`: مسیر پوشه‌ای که تصاویر داخل آن هستند.
  - `max_workers`: حداکثر تعداد *thread*هایی که همزمان می‌توانند کار کنند (پیش‌فرض ۸).
- **جمع‌آوری مسیر تصاویر:**
  - با `os.listdir(image_folder)` تمام فایل‌های داخل پوشه گرفته می‌شوند.
  - با شرط `endswith` فقط فایل‌هایی که پسوند `.jpg`، `.jpeg` یا `.png` دارند انتخاب می‌شوند.

- نتیجه یک لیست از نام فایل‌ها است که به `images` ذخیره می‌شود.

- **پردازش موازی با `ThreadPoolExecutor`:**
  - `ThreadPoolExecutor` یک *thread pool* ایجاد می‌کند تا چند تصویر همزمان پردازش شوند.
  - *executor.submit(func, arg1, arg2, ...)* یک کار (*task*) برای اجرای تابع `func` با آرگومان‌های مشخص ایجاد می‌کند.
  - برای هر تصویر:
    - نام فایل بدون پسوند `(os.path.splitext(os.path.basename(img_name))[0]` به عنوان `item_id` استفاده می‌شود.
    - مسیر تصویر به `store_image_item` داده می‌شود تا *embedding* آن ذخیره شود.
  - نتیجه یک لیست از `futures` است. هر `future` نماینده نتیجه یک کار در حال اجرا است.

- **نمایش پیشرفت پردازش:**
  - `as_completed(futures)` یک *generator* برمی‌گرداند که هر بار یکی از کارها تمام شود، آن را yield می‌کند.
  - *tqdm(..., desc="Processing images")* یک نوار پیشرفت (*progress bar*) نمایش می‌دهد.

تابع `store_image_item`

در تابع *process_images* از تابع زیر استفاده شده است

```python
def store_image_item(item_id: str, img_name: str):
    """Store image embedding and metadata in Weaviate."""
    embedding = get_embedding("image", img_name)
    relative_path = f"/content/image_dataset/{img_name}"
    properties = {
        "contentId": item_id,
        "modality": "image",
        "filePath": relative_path,
        "content": "",
    }
    collection.data.insert(properties=properties, vector=embedding.tolist())
```

توضیح پارامترها و مراحل:

- **پارامترها:**
  - `item_id`: شناسه یکتا برای هر تصویر (معمولاً نام فایل بدون پسوند).
  - `img_name`: نام فایل تصویر یا مسیر نسبی آن.

- **تولید embedding تصویر:**
  - با استفاده از تابع *get_embedding("image", img_name)*، تصویر به یک بردار عددی (vector) تبدیل می‌شود.
  - این embedding برای جستجوی شباهت و یادگیری ماشین استفاده می‌شود.

- **تعیین مسیر نسبی تصویر:**
  - `relative_path = f"/content/image_dataset/{img_name}"` برای ذخیره در metadata ساخته می‌شود.

- **آماده‌سازی metadata:**
  - `properties` شامل ویژگی‌های تصویر است:
    - `contentId`: شناسه یکتا تصویر
    - `modality`: نوع داده، اینجا `"image"`
    - `filePath`: مسیر تصویر
    - `content`: خالی برای تصاویر

- **ذخیره در Weaviate:**
  - `collection.data.insert(properties=properties, vector=embedding.tolist())` داده‌ها و embedding تصویر را در دیتابیس ذخیره می‌کند.
  - این کار تصویر و بردار ویژگی آن را برای جستجوی مشابهت و بازیابی سریع آماده می‌کند.

تابع `get_embedding`

برای تولید embedding برای تصاویر از تابع زیر استفاده شده است

```python
def get_embedding(modality: str, image_name: Union[str, None]) -> np.ndarray:
    mod = modality.lower()
    if mod == "image":
        if not isinstance(image_name, str):
            raise ValueError("`image_name` must be a string.")

        img_path = f"./content/image_dataset/{image_name}"
        img = Image.open(img_path).convert("RGB")
        x = preprocess(img).unsqueeze(0).to(DEVICE)

        with torch.no_grad():
            features = clip_model.encode_image(x)
            features = features / features.norm(dim=-1, keepdim=True)
        return features.detach().cpu().numpy().reshape(-1)
    else:
        raise ValueError("`modality` must be 'image'.")
```

توضیح تابع:

- **ورودی‌ها:**
  - `modality`: نوع داده
  - `image_name`: نام فایل تصویر (یا None اگر ورودی نامعتبر باشد).
- **بارگذاری و آماده‌سازی تصویر:**
  - مسیر تصویر ساخته می‌شود: `img_path = f"./content/image_dataset/{image_name}"`.
  - تصویر با PIL باز و به RGB تبدیل می‌شود.
  - با تابع `preprocess` آماده‌سازی می‌شود (تغییر اندازه، نرمال‌سازی و تبدیل به *Tensor*).
  - `unsqueeze(0)` برای اضافه کردن بعد batch و `.to(DEVICE)` برای انتقال به CPU/GPU.
- **تولید embedding با CLIP:**
  - با `torch.no_grad()` محاسبه گرادیان غیرفعال می‌شود (صرفه‌جویی در حافظه و سرعت).
  - `clip_model.encode_image(x)` تصویر را به یک بردار ویژگی (embedding) تبدیل می‌کند.
  - نرمال‌سازی: *features / features.norm(dim=-1,keepdim=True)* تا طول بردار برابر ۱ شود (مناسب جستجوی شباهت *cosine*).
- **تبدیل به NumPy و بازگرداندن:**
  - `.detach()` از گراف محاسباتی جدا می‌کند.
  - `.cpu()` بردار را روی CPU قرار می‌دهد.
  - `.numpy()` به آرایه NumPy تبدیل می‌کند.
  - `.reshape(-1)` بردار را یک‌بعدی می‌کند و باز می‌گرداند.

۲.۶ وارد کردن داده‌های صوتی به دیتابیس

در این بخش، فرآیند وارد کردن داده‌های صوتی به پایگاه داده توضیح داده می‌شود. در ابتدا، پس از قرار دادن تمامی فایل‌های صوتی در پوشه‌ی مربوطه، یک پنجره‌ی *cmd* باز کرده و به مسیر پروژه بروید. سپس با استفاده از دستور زیر، اسکریپت پایتون با نام *audio.py* اجرا می‌شود:

```
python import_data_into_database/import_local_data/audio.py
```

این اسکریپت تمام فایل‌های صوتی داخل پوشه *PROECT_ROOT/content/audio_dataset* را خوانده و با استفاده از مدل Whisper متن آنها را به دست می‌آورد و همچنین متن‌های غیرانگلیسی را به فارسی ترجمه میکند. سپس برای متن‌های به دست آمده با استفاده از مدل CLIP یک embedding تولید میکند و در نهایت این embedding را به همراه مسیر فایل صوتی و همچنین متن به دست آمده در دیتابیس ذخیره میکند.

۱.۲.۶ توضیح کد

در ابتدای اجرای این اسکریپت، اتصال به پایگاه داده‌ی *Weaviate* و کالکشن (*collection*) از پیش ساخته‌شده برقرار می‌شود. این کار توسط کد زیر انجام می‌گیرد:

```
try:
    weaviate_client = weaviate.connect_to_local()
    print("  Connected to Weaviate")
    collection = weaviate_client.collections.get("Multimodal_Collection")
except Exception as e:
    print(f"  Weaviate connection failed: {e}")
    weaviate_client = None
```

در این بخش، تابع *connect_to_local()* اتصال به سرویس محلی *Weaviate* را برقرار می‌کند (همان سرویسی که با *Docker* روی *localhost:8080* اجرا شده است).

۲.۲.۶ تنظیمات مدل *CLIP*

در ادامه، تنظیمات مربوط به مدل *CLIP* انجام می‌شود. این مدل برای تولید *embedding* داده‌ها به‌کار می‌رود:

```
MODEL_NAME = "ViT-B-32"
PRETRAINED_LOCAL_PATH = (
    "./open_clip_weights/ViT-B-32-openai/open_clip_model.safetensors"
)
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"

clip_model, _, preprocess = open_clip.create_model_and_transforms(
    MODEL_NAME, pretrained=PRETRAINED_LOCAL_PATH
)
tokenizer = open_clip.get_tokenizer(MODEL_NAME)
clip_model.to(DEVICE).eval()
```

- *MODEL_NAME*: نام معماری مدل *CLIP* را مشخص می‌کند (مدل *Vision Transformer Base* با اندازه‌ی پچ ۳۲×۳۲).
- *PRETRAINED_LOCAL_PATH*: مسیر فایل وزن‌های ازپیش‌آموزش‌داده‌شده‌ی مدل را تعیین می‌کند تا از آن بارگذاری شود.
- *DEVICE*: بررسی می‌کند که آیا *GPU (CUDA)* در دسترس است یا خیر؛ در صورت وجود از *GPU* و در غیر این صورت از *CPU* استفاده می‌شود.
- *create_model_and_transforms*: مدل *CLIP* و توابع پیش‌پردازش آن را بارگذاری می‌کند.
- *tokenizer*: توکنایزر مخصوص مدل را دریافت می‌کند تا متون به توکن‌های عددی تبدیل شوند.
- *clip_model.to(DEVICE).eval()*: مدل را به دستگاه مربوطه منتقل کرده و در حالت ارزیابی قرار می‌دهد.

۳.۲.۶ مدل *Whisper*

در این بخش از مدل *Whisper* برای استخراج متن از فایل‌های صوتی استفاده می‌شود:

```
MODEL_TYPE = "small"
whisper_model = whisper.load_model(MODEL_TYPE, device=DEVICE)
```

مدل *Whisper* به کمک پارامتر *MODEL_TYPE* مقداردهی می‌شود. این پارامتر می‌تواند یکی از مقادیر *small*، *medium* یا *large* باشد. مدل‌های *medium* و *large* تنها بر روی *GPU* قابل اجرا هستند، در حالی‌که مدل *small* بر روی *CPU* نیز قابل بارگذاری است، هرچند دقت پایین‌تری دارد. همچنین، مدل *small* برای فایل‌های صوتی زبان انگلیسی عملکرد مناسبی دارد ولی برای زبان فارسی ضعیف‌تر است؛ بنابراین در صورت نیاز به پشتیبانی از زبان فارسی، استفاده از مدل‌های *medium* یا *large* توصیه می‌شود.

۴.۲.۶ بارگذاری و پردازش داده‌های صوتی

در ادامه، داده‌های صوتی بارگذاری و پردازش می‌شوند:

```python
if __name__ == "__main__":
    audio_folder = "./content/audio_dataset"
    audio_json = f"{audio_folder}/audio_metadata.json"

    transcribe_audios(audio_folder, audio_json)
    process_audio_json(audio_json)
    weaviate_client.close()
```

مسیر فایل‌ها به‌صورت خودکار تنظیم می‌شود و نیازی به تغییر ندارد. پردازش داده‌های صوتی در دو مرحله انجام می‌شود:

۱. استخراج متن (*Transcription*) از فایل‌های صوتی؛

۲. تولید *embedding* از متون و ذخیره در پایگاه داده.

۵.۲.۶ مرحله اول: استخراج متن از فایل‌های صوتی

این کار با تابع زیر انجام می‌شود:

```python
def transcribe_audios(folder, out_json="results.json"):
    if not os.path.isdir(folder):
        raise ValueError(f"No folder: {folder}")

    files = sorted(
        [
            os.path.join(folder, file_path)
            for file_path in os.listdir(folder)
            if file_path.lower().endswith((".wav", ".mp3", ".m4a"))
        ]
    )

    results = [
        {
            "id": i + 1,
            "filename": os.path.basename(audio_path),
            "audio_path": audio_path,
            "transcription": transcribe_to_english(audio_path),
        }
        for i, audio_path in enumerate(tqdm(files, desc="Processing"))
    ]

    with open(out_json, "w", encoding="utf-8") as f:
        json.dump(results, f, ensure_ascii=False, indent=2)
    print(f" Saved {len(results)} items to {out_json}")
    return results
```

تعریف تابع و ورودی‌ها

```python
def transcribe_audios(folder, out_json="results.json"):
```

- *folder*: مسیر پوشه‌ای که فایل‌های صوتی در آن قرار دارند.
- *out_json*: نام فایل خروجی با فرمت *JSON* (به‌صورت پیش‌فرض برابر با *"results.json"*).

جمع‌آوری فایل‌های صوتی

در گام نخست، تابع تمامی فایل‌های صوتی موجود در پوشه‌ی مشخص‌شده را شناسایی و آماده‌ی پردازش می‌کند.

```python
files = sorted(
    [
        os.path.join(folder, file_path)
        for file_path in os.listdir(folder)
        if file_path.lower().endswith((".wav", ".mp3", ".m4a"))
    ]
)
```

در این بخش:

- تمامی فایل‌های موجود در پوشه بررسی می‌شوند.
- تنها فایل‌هایی که پسوند آن‌ها *.wav*، *.mp3* یا *.m4a* باشد انتخاب می‌گردند.

- مسیر کامل هر فایل با استفاده از تابع *os.path.join* ساخته می‌شود.
- سپس فهرست فایل‌ها با دستور *sorted* مرتب می‌شود.

در نهایت، متغیر *files* شامل فهرستی از مسیر کامل تمام فایل‌های صوتی موجود در پوشه است.

ایجاد خروجی برای هر فایل

در این مرحله، برای هر فایل صوتی موجود در فهرست، خروجی متنی (متن استخراج‌شده از گفتار) تولید و در قالب یک ساختار داده‌ی دیکشنری ذخیره می‌گردد.

```
results = [
    {
        "id": i + 1,
        "filename": os.path.basename(audio_path),
        "audio_path": audio_path,
        "transcription": transcribe_to_english(audio_path),
    }
    for i, audio_path in enumerate(tqdm(files, desc="Processing"))
]
```

در این بخش:

- از کتابخانه‌ی *tqdm* برای نمایش نوار پیشرفت استفاده می‌شود.
- برای هر فایل صوتی:
  - *id*: شماره‌ی ترتیبی فایل (از عدد ۱ آغاز می‌شود).
  - *filename*: نام فایل بدون مسیر.
  - *audio_path*: مسیر کامل فایل.
  - *transcription*: نتیجه‌ی حاصل از اجرای تابع *transcribe_to_english(audio_path)* که متن انگلیسی استخراج‌شده از گفتار در فایل است.

در پایان، خروجی این بخش شامل فهرستی از دیکشنری‌هاست که هر کدام نمایانگر اطلاعات و متن استخراج‌شده از یک فایل صوتی می‌باشد.

ذخیره در فایل *JSON*

در این مرحله، داده‌های استخراج‌شده در قالب فایل *JSON* ذخیره می‌شوند.

```
with open(out_json, "w", encoding="utf-8") as f:
    json.dump(results, f, ensure_ascii=False, indent=2)
```

توضیحات:

- فایل خروجی *JSON* باز می‌شود و محتوای نتایج در آن نوشته می‌شود.
- پارامتر *ensure_ascii=False* تضمین می‌کند که کاراکترهای غیرانگلیسی (مانند فارسی) به‌درستی ذخیره شوند.
- مقدار *indent=2* برای زیبایی و خوانایی ساختار فایل خروجی استفاده می‌گردد.

تابع *transcribe_to_english*

برای استخراج متن از هر فایل صوتی، از تابع *transcribe_to_english* استفاده می‌شود. این تابع وظیفه دارد یک فایل صوتی را دریافت کرده و متن آن را به زبان انگلیسی برگرداند، صرف‌نظر از اینکه زبان اصلی گفتار چه باشد.

```
def transcribe_to_english(path):
    res = whisper_model.transcribe(path, task="transcribe")
    lang, text = res.get("language", ""), res.get("text", "").strip()
    if lang.startswith("en"):
        return text
    else:
        return (
            whisper_model.transcribe(path, task="translate", language="en")
            .get("text", "")
            .strip()
        )
```

ورودی

- *path*: مسیر فایل صوتی ورودی .

استخراج گفتار (*Transcription*)

```
res = whisper_model.transcribe(path, task="transcribe")
```

در این مرحله از مدل *Whisper* (مدل تشخیص گفتار از *OpenAI*) استفاده می‌شود. پارامتر *task="transcribe"* به مدل اعلام می‌کند که تنها گفتار را به همان زبان اصلی به متن تبدیل کند، بدون ترجمه. خروجی حاصل در متغیر *res* ذخیره می‌شود که ساختاری مشابه زیر دارد:

```
{
    "text": "Bonjour tout le monde",
    "language": "fr"
}
```

استخراج زبان و متن

```
lang, text = res.get("language", ""), res.get("text", "").strip()
```

در این مرحله:

- *lang*: زبان تشخیص‌داده‌شده توسط مدل (مانند *"en"*، *"fr"*، *"fa"*، *"de"* و ...).
- *text*: متن استخراج‌شده از گفتار (پس از حذف فاصله‌های اضافی).

بررسی زبان انگلیسی

```
if lang.startswith("en"):
    return text
```

اگر زبان تشخیص‌داده‌شده انگلیسی باشد (یعنی مقدار *lang* با *"en"* شروع شود)، همان متن اولیه مستقیماً بازگردانده می‌شود.

ترجمه در صورت غیرانگلیسی بودن زبان

```
else:
    return (
        whisper_model.transcribe(path, task="translate", language="en")
        .get("text", "")
        .strip()
    )
```

در صورتی که زبان اصلی غیرانگلیسی باشد:

- مدل *Whisper* مجدداً اجرا می‌شود، اما این بار با پارامترهای:
  - *task="translate"*: به این معنا که مدل همزمان با تشخیص گفتار، ترجمه به انگلیسی را نیز انجام دهد.
  - *language="en"*: خروجی نهایی حتماً به زبان انگلیسی باشد.
- در انتها، تنها مقدار مربوط به کلید *"text"* استخراج شده و فاصله‌های اضافی آن حذف می‌شود.

خروجی نهایی

تابع همواره یک رشته متنی (*string*) بازمی‌گرداند که نسخه‌ی انگلیسی گفتار موجود در فایل صوتی است، صرف‌نظر از اینکه زبان اصلی صوت چه بوده است.

در پایان این مرحله، یک فایل *JSON* شامل فراداده‌ها (*metadata*) و متن‌های استخراج‌شده در همان پوشه‌ای که فایل‌های صوتی قرار دارند تولید می‌شود. به این ترتیب، برای هر داده‌ی صوتی، یک متن معادل انگلیسی نیز در اختیار خواهیم داشت.

۶.۲.۶ مرحله دوم: تولید و ذخیره‌ی *embedding*

در این مرحله، با استفاده از تابع *process_audio_json*، فایل *JSON* مرحله‌ی قبل خوانده می‌شود و برای هر داده صوتی *embedding* تولید می‌گردد:

```
def process_audio_json(json_path: str, max_workers: int = 8):
    with open(json_path, "r") as f:
        metadata = json.load(f)

    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        futures = []
        for item in metadata:
            file_path = item.get("audio_path", "").strip()
            transcription = item.get("transcription", "").strip()
            if not (file_path and transcription):
```

```
                continue
            content_id = str(item.get("id", os.path.basename(file_path)))
            futures.append(
                executor.submit(
                    store_audio_item,
                    content_id,
                    transcription,
                    os.path.basename(file_path),
                )
            )

        for _ in tqdm(
            as_completed(futures), total=len(futures), desc="  Processing audios"
        ):
            pass

    print(f"  Processed {len(metadata)} audio files.")
```

تعریف تابع

```
def process_audio_json(json_path: str, max_workers: int = 8):
"""Process and store audio embeddings using metadata JSON."""
```

ورودی ها:

- *json_path*: مسیر فایل *JSON* شامل متادیتا (اطلاعات مربوط به فایل‌های صوتی و متن آن‌ها).
- *max_workers*: تعداد نخ‌ها (*threads*) برای اجرای موازی (پیش‌فرض: ۸).

خواندن فایل *JSON* ورودی

```
with open(json_path, "r") as f:
    metadata = json.load(f)
```

فایل *JSON* باز شده و محتویات آن در متغیر *metadata* ذخیره می‌شود.

ایجاد *ThreadPool* برای پردازش موازی

```
with ThreadPoolExecutor(max_workers=max_workers) as executor:
```

از *ThreadPoolExecutor* برای اجرای هم‌زمان چند پردازش استفاده می‌شود. تعداد نخ‌ها برابر مقدار *max_workers* تعیین می‌شود.

آماده‌سازی وظایف (Tasks)

```
futures = []
for item in metadata:
    file_path = item.get("audio_path", "").strip()
    transcription = item.get("transcription", "").strip()
    if not (file_path and transcription):
        continue
```

در این بخش: مسیر فایل صوتی (*audio_path*) و متن مربوطه (*transcription*) گرفته می‌شود. اگر مسیر یا متن خالی باشد، آن مورد نادیده گرفته می‌شود.

ارسال کارها به *ThreadPool*

```
content_id = str(item.get("id", os.path.basename(file_path)))
futures.append(
    executor.submit(
        store_audio_item,
        content_id,
        transcription,
        os.path.basename(file_path),
    )
)
```

برای هر فایل معتبر، تابع *store_audio_item* به‌صورت غیرهم‌زمان اجرا می‌شود.
ورودی‌های آن شامل موارد زیر است:

- *content_id*: شناسه‌ی فایل (از *id* یا نام فایل).
- *transcription*: متن گفتار استخراج‌شده.
- *os.path.basename(file_path)*: نام فایل بدون مسیر.

نمایش نوار پیشرفت

```
for _ in tqdm(
    as_completed(futures), total=len(futures), desc="  Processing audios"
):
    pass
```

از *tqdm* برای نمایش نوار پیشرفت استفاده می‌شود. *as_completed(futures)* وضعیت تکمیل هر نخ را پیگیری می‌کند.
خروجی نهایی در پایان، پیام موفقیت چاپ می‌شود: *"Processed X audio files."* که نشان می‌دهد چند فایل پردازش و ذخیره شده‌اند.

تابع *store_audio_item*
در تابع بالا از تابع زیر استفاده شده است. این تابع وظیفه دارد اطلاعات مربوط به هر فایل صوتی (شناسه، متن استخراج‌شده و *embedding*) را در پایگاه داده‌ی برداری *Weaviate* ذخیره نماید. در این فرآیند، بردار *embedding* از متن استخراج‌شده (و نه از خود فایل صوتی) تولید می‌شود.

```
def store_audio_item(item_id: str, transcription_text: str, audio_name: str = ""):
    """Store audio transcription embedding and metadata in Weaviate."""
    relative_path = f"/content/audio_dataset/{audio_name}"
    embedding = get_embedding("text", transcription_text)
    properties = {
        "contentId": item_id,
        "modality": "audio",
        "filePath": relative_path,
        "content": transcription_text,
    }
    collection.data.insert(properties=properties, vector=embedding.tolist())
```

تعریف تابع

```
def store_audio_item(item_id: str, transcription_text: str, audio_name: str = ""):
"""Store audio transcription embedding and metadata in Weaviate."""
```

ورودی‌ها:

- *item_id*: شناسه‌ی منحصربه‌فرد فایل (می‌تواند از فایل *JSON* یا نام فایل گرفته شود).
- *transcription_text*: متنی که از گفتار صوتی استخراج شده است.

ساخت مسیر فایل (نسبی)

```
relative_path = f"/content/audio_dataset/{audio_name}"
```

تولید *embedding* **از متن گفتار**

```
embedding = get_embedding("text", transcription_text)
```

- تابع *get_embedding* یک بردار عددی (Embedding) از محتوای معنایی متن تولید می‌کند.
- پارامتر *"text"* نشان می‌دهد که نوع داده‌ی ورودی، متن است (نه تصویر یا صدا).
- این بردار برای متن استخراج‌شده تولید می‌شود، نه خود فایل صوتی.
- بردار *embedding* در مراحل بعدی برای جستجوی معنایی (*semantic search*) در *Weaviate* مورد استفاده قرار می‌گیرد.

آماده‌سازی متادیتا برای ذخیره

```
properties = {
    "contentId": item_id,
    "modality": "audio",
    "filePath": relative_path,
    "content": transcription_text,
}
```

- یک دیکشنری از ویژگی‌های شیء ساخته می‌شود که شامل اطلاعات توصیفی فایل است:
  - *"contentId"*: شناسه‌ی منحصربه‌فرد فایل.
  - *"modality"*: نوع داده

- *"filePath"*: مسیر فایل صوتی در دیتاست.
- *"content"*: متن استخراج‌شده از گفتار.

درج داده در پایگاه داده‌ی *Weaviate*

```
collection.data.insert(properties=properties, vector=embedding.tolist())
```

- داده‌ها در یک *Collection* (برای مثال *"Multimodal_Collection"*) در *Weaviate* ذخیره می‌شوند.
- پارامترها شامل موارد زیر هستند:
  - *properties*: متادیتا و اطلاعات توصیفی شامل شناسه، مسیر، نوع داده و متن.
  - *vector*: بردار عددی *embedding* که به صورت لیست ذخیره می‌شود.
- به این ترتیب، هر داده‌ی صوتی در *Weaviate* شامل دو بخش اصلی است:
  - متادیتا (ویژگی‌های توصیفی)
  - بردار معنایی (*embedding*)
- این ساختار باعث می‌شود سیستم بتواند در مراحل بعدی بر اساس معنا و شباهت محتوایی میان فایل‌های صوتی، عملیات جست‌وجو و بازیابی مؤثر انجام دهد.

تابع *get_embedding*

```
def get_embedding(modality: str, input_data: Union[str, None]) -> np.ndarray:
    mod = modality.lower()
    if mod == "text":
        if not isinstance(input_data, str):
            raise ValueError("`input_data` must be a string.")

        tokens = tokenizer([input_data]).to(DEVICE)
        with torch.no_grad():
            features = clip_model.encode_text(tokens)
            features = features / features.norm(dim=-1, keepdim=True)
        return features.detach().cpu().numpy().reshape(-1)
    else:
        raise ValueError("`modality` must be 'text'.")
```

در تابع *store_audio_item* از تابع زیر برای به دست آوردن embedding استفاده شده است. این تابع یک بردار *embedding* نرمال‌شده بر اساس نوع داده تولید می‌کند.

تعریف تابع و پارامترها

```
def get_embedding(modality: str, input_data: Union[str, None]) -> np.ndarray:
```

- *modality*: نوع داده
- *input_data*: داده‌ی ورودی برای تولید *embedding* (رشته‌ی متنی).
- خروجی: آرایه‌ی *NumPy* (*np.ndarray*) که بردار عددی متن را باز می‌گرداند.

توکنایز کردن متن

```
tokens = tokenizer([input_data]).to(DEVICE)
```

- متن با tokenizer مدل *CLIP* به توکن تبدیل می‌شود.
- توکن‌ها به دستگاه مناسب (*CPU* یا *GPU*) منتقل می‌شوند.

تولید *embedding* با مدل CLIP

```
with torch.no_grad():
    features = clip_model.encode_text(tokens)
    features = features / features.norm(dim=-1, keepdim=True)
```

- *torch.no_grad()*: جلوگیری از ذخیره‌سازی گرادیان‌ها برای صرفه‌جویی در حافظه و زمان.

- *clip_model.encode_text(tokens)*: تولید بردار عددی متن *(text embedding)* با مدل *CLIP*.
- *features / features.norm(dim=-1,keepdim=True)*: نرمال‌سازی بردار به طول یک.

بازگرداندن خروجی به *NumPy*

```
return features.detach().cpu().numpy().reshape(-1)
```

- *detach()*: جدا کردن بردار از گراف محاسباتی .PyTorch
- *cpu()*: انتقال داده به حافظه .CPU
- *numpy()*: تبدیل به آرایه‌ی *NumPy*.
- *reshape(-1)*: تبدیل به بردار یک‌بعدی.

۳.۶ فرآیند وارد کردن داده‌های متنی

به منظور وارد کردن داده‌های متنی به دیتابیس، لازم است متون مورد نظر در یک فایل با فرمت *CSV* ذخیره شده و در پوشه مربوطه قرار گیرد. در ادامه، مراحل اجرای اسکریپت و توضیح جزئیات کد ارائه شده است.

۱.۳.۶ اجرای اسکریپت

برای اجرای اسکریپت پایتون، ابتدا یک خط فرمان (*CMD*) را باز کرده و به مسیر ریشه‌ی پروژه بروید. سپس، با استفاده از دستور زیر، اسکریپت پایتون با نام *texts.py* را اجرا کنید:

```
python import_data_into_database/import_local_data/texts.py
```

۲.۳.۶ توضیحات جزئیات کد

این اسکریپت، وظیفه‌ی اتصال به دیتابیس، بارگذاری مدل *CLIP*، تعریف زنجیره‌ی ترجمه، پردازش داده‌ها (شامل ترجمه و تولید *embedding*) و ذخیره‌ی نهایی در دیتابیس *Weaviate* را بر عهده دارد.

۱. اتصال به دیتابیس *Weaviate* این بخش از کد، یک اتصال به سرویس محلی *Weaviate* که با *Docker* روی *localhost:8080* اجرا شده است، برقرار می‌سازد و سپس شیء *collection* مورد نظر را بازیابی می‌کند.

```
try:
    weaviate_client = weaviate.connect_to_local()
    print(" Connected to Weaviate")
    collection = weaviate_client.collections.get("Multimodal_Collection")
except Exception as e:
    print(f" Weaviate connection failed: {e}")
    weaviate_client = None
```

۲. تنظیمات مدل *CLIP* برای تولید *Embedding* کد زیر تنظیمات مربوط به مدل *CLIP* که برای تولید بردارهای عددی (*embedding*) داده‌های متنی و تصویری به کار می‌رود را انجام می‌دهد.

```
MODEL_NAME = "ViT-B-32"
PRETRAINED_LOCAL_PATH = (
    "./open_clip_weights/ViT-B-32-openai/open_clip_model.safetensors"
)
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
clip_model, _, preprocess = open_clip.create_model_and_transforms(
    MODEL_NAME, pretrained=PRETRAINED_LOCAL_PATH
)
tokenizer = open_clip.get_tokenizer(MODEL_NAME)
clip_model.to(DEVICE).eval()
```

توضیح پارامترها:

- *MODEL_NAME = "ViT-B-32"*: نام معماری مدل *CLIP* (مدل *Vision Transformer Base* با اندازه‌ی *patch* برابر با 32×32) مشخص می‌شود.
- *PRETRAINED_LOCAL_PATH*: مسیر فایل وزن‌های از پیش آموزش داده شده‌ی مدل را تعیین می‌کند.

- *DEVICE = "cuda" if torch.cuda.is_available() else "cpu"*: در دسترس بودن *GPU* (*CUDA*) بررسی می‌شود. اگر *GPU* در دسترس باشد از آن، در غیر این صورت از *CPU* استفاده خواهد شد.
- *open_clip.create_model_and_transforms*: مدل *CLIP* را از مسیر داده‌شده بارگذاری کرده و تابع پیش‌پردازش تصاویر (*preprocess*) را برمی‌گرداند.
- *open_clip.get_tokenizer*: توکنایزر مخصوص همان مدل برای تبدیل متون به توکن‌های عددی قابل پردازش توسط مدل را بازیابی می‌کند.
- *clip_model.to(DEVICE).eval()*: مدل را روی دستگاه تعیین شده منتقل کرده و در حالت ارزیابی (*eval*) قرار می‌دهد تا برای پیش‌بینی آماده شود.

۳. بارگذاری کلید *API* برای ترجمه از آنجا که داده‌های متنی فارسی باید توسط یک مدل زبانی بزرگ (*LLM*) ترجمه شوند، نیاز است کلید *API* دریافتی از *OpenRouter* که در فایل *".env"* ذخیره شده است، بارگذاری شود.

```
load_dotenv()
API_KEY = os.getenv("LLM_API_KEY")
if not API_KEY:
    raise ValueError(
        "Please set your API key in the environment variable OPENROUTER_API_KEY."
    )
```

توضیح:

- *load_dotenv()*: این دستور فایل *env.* را خوانده و تمام متغیرهای محیطی (*environment variables*) داخل آن را در محیط اجرایی پایتون بارگذاری می‌کند.
- *API_KEY = os.getenv("LLM_API_KEY")*: با استفاده از تابع *os.getenv()*، مقدار متغیر محیطی با نام *"LLM_API_KEY"* خوانده و در متغیر *API_KEY* ذخیره می‌شود.

۴. تعریف زنجیره‌ی پردازش (*Pipeline*) برای ترجمه کد زیر یک زنجیره‌ی پردازش هوشمند برای ترجمه‌ی خودکار متن فارسی به انگلیسی ایجاد می‌کند که شامل مراحل دریافت متن فارسی، آماده‌سازی *prompt* ترجمه، ارسال به مدل *GPT* از طریق *OpenRouter* و دریافت ترجمه‌ی خالص است.

```
LLM_model = ChatOpenAI(
    model="openai/gpt-oss-20b:free",
    temperature=0.1,
    openai_api_base="https://openrouter.ai/api/v1",
    openai_api_key=API_KEY,
)
prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            "Translate the following Persian (Farsi) text to clear, natural English.\n"
            "Return only the translation (no extra explanation).\n\n"
            "Persian text:\n{text}",
        )
    ]
)
parser = StrOutputParser()
vision_chain = prompt | LLM_model | parser
```

تشریح اجزای زنجیره‌ی ترجمه:

۱. **ساخت مدل زبانی (*LLM*)**

```
LLM_model = ChatOpenAI(
    model="openai/gpt-oss-20b:free",
    temperature=0.1,
    openai_api_base="https://openrouter.ai/api/v1",
    openai_api_key=API_KEY,
)

```

- **کارکرد:** ایجاد یک شیء مدل زبانی (*LLM*) از کلاس *ChatOpenAI* (کتابخانه‌ی *LangChain*) برای برقراری ارتباط با مدل *GPT* از طریق *API*.

- **پارامترها:**
  (آ) *model="openai/gpt-oss-20b:free"*: نام مدل زبانی که از *OpenRouter* استفاده می‌کند
  (ب) *temperature=0.1*: درجه‌ی تصادفی بودن پاسخ‌ها را مشخص می‌کند (هرچه کمتر، پاسخ‌ها دقیق‌تر و تکراری‌تر).
  (ج) *openai_api_base="https://openrouter.ai/api/v1"*: آدرس پایه‌ی *API* سرویس *OpenRouter* (جایگزین مستقیم *OpenAI API*).
  (د) *openai_api_key=API_KEY*: کلید *API* شما برای احراز هویت در *OpenRouter*.

۲. **تعریف قالب پیام** (*prompt template*)

```
prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            "Translate the following Persian (Farsi) text to clear, natural English.\n"
            "Return only the translation (no extra explanation).\n\n"
            "Persian text:\n{text}",
        )
    ]
)

```

- **کارکرد:** ساخت یک الگو برای پیام ورودی که به مدل داده می‌شود.
- **توضیح جزئی:**
  (آ) *"system"*: نوع پیام (به مدل دستور می‌دهد چگونه رفتار کند).
  (ب) پیام داخل کوتیشن: دستور ترجمه است؛ از مدل خواسته شده: متن فارسی را ترجمه کند، فقط ترجمه را برگرداند (بدون توضیح اضافه).
  (ج) *{text}*: متغیری است که هنگام اجرا با متن واقعی جایگزین می‌شود.

۳. **ساخت مفسر خروجی** (*parser*)

```
parser = StrOutputParser()

```

- **کارکرد:** این مفسر خروجی خام مدل را گرفته و به رشته‌ی متنی نهایی (*string*) تبدیل می‌کند. به‌عبارتی، خروجی *JSON* یا ساختاریافته‌ی مدل را ساده کرده و فقط متن ترجمه را برمی‌گرداند.

۴. **ساخت زنجیره‌ی پردازش** (*chain*)

```
vision_chain = prompt | LLM_model | parser

```

- **کارکرد:** با استفاده از اپراتور *pipe* (|) در *LangChain*، سه جزء را به‌صورت مرحله‌به‌مرحله به هم وصل می‌کند: *prompt* (قالب ورودی)، *LLM_model* (مدل زبانی)، و *parser* (مفسر خروجی).
- **نتیجه:** *vision_chain* یک زنجیره‌ی کامل است که با دریافت یک متن فارسی، ترجمه‌ی انگلیسیِ نهایی را برمی‌گرداند.

۵. **بارگذاری و پردازش داده‌های متنی** بخش اصلی اسکریپت که فایل ورودی *CSV* را مشخص کرده و فرآیند دو مرحله‌ای پردازش را آغاز می‌کند:

```
if __name__ == "__main__":
    # Step 1: Input file (contains Persian + English)
    input_csv = "./content/text_dataset/text_data.csv"

    # Step 2: Prepare final English-only CSV
    final_csv = prepare_final_csv(input_csv, text_column="text", workers=2)

    # Step 3: Generate and store embeddings in Weaviate
    process_texts(final_csv)
    weaviate_client.close()
```

فرآیند پردازش داده‌های متنی در دو مرحله انجام می‌شود:

۱. **مرحله اول: ترجمه متون به انگلیسی:** تمام داده‌های متنی (فارسی و انگلیسی) پردازش شده و تمام داده‌ها به انگلیسی ترجمه می‌شوند.

۲. **مرحله دوم: تولید *embedding* و ذخیره در دیتابیس:** برای داده‌های انگلیسی تولید شده، *embedding* محاسبه شده و در دیتابیس *Weaviate* ذخیره می‌شود.

مرحله اول: ترجمه متون به انگلیسی با استفاده از تابع *prepare_final_csv* تمام داده‌ها را به انگلیسی ترجمه می‌کنیم. اگر متن انگلیسی باشد نیاز به ترجمه نیست، اما اگر فارسی باشد ابتدا با استفاده از *LLM* آن را ترجمه می‌کنیم.
تابع *prepare_final_csv()* سه ورودی (پارامتر) می‌گیرد:

۱. *input_csv*: مسیر فایل *CSV* ورودی (حاوی متن‌ها، فارسی و انگلیسی).

۲. *text_column="text"*: نام ستونی از *CSV* که متن‌ها در آن قرار دارند.

۳. *workers=2*: تعداد (*threads*) یا پردازنده‌های موازی برای سرعت‌دهی به پردازش.

خروجی این تابع یک فایل *CSV* جدید خواهد بود که در آن تمام داده‌ها به انگلیسی ترجمه شده‌اند.

توضیح تابع *prepare_final_csv* این تابع با استفاده از *LLM* متون فارسی را ترجمه می‌کند و در نهایت یک فایل *CSV* جدید حاوی تمام متون ترجمه شده به زبان انگلیسی تولید می‌کند.

```python
def prepare_final_csv(
    input_csv: str, text_column="text", translated_column="translated", workers=2
):
    """
    Process a CSV file where each row can be Persian or English.
    - English rows are added directly to final.csv
    - Persian rows are translated and also added to the same 'text' column
    """
    if not os.path.isfile(input_csv):
        raise FileNotFoundError(f"File '{input_csv}' does not exist.")

    df = pd.read_csv(input_csv)

    if text_column not in df.columns:
        raise ValueError(
            f"Column '{text_column}' does not exist in the file. Columns:
    {df.columns.tolist()}"
        )

    # Split English and Persian rows
    english_rows = df[~df[text_column].apply(is_persian)]
    persian_rows = df[df[text_column].apply(is_persian)]

    # Paths
    folder, filename = os.path.split(input_csv)
    name, ext = os.path.splitext(filename)
    final_csv = os.path.join(folder, f"{name}_final.csv")
    temp_csv = os.path.join(folder, f"{name}_temp.csv")

    # Save English rows directly to final.csv
    english_rows[[text_column]].to_csv(final_csv, index=False, encoding="utf-8-sig")

    if persian_rows.empty:
        print("No Persian text found. Final file saved.")
        return final_csv

    # Save Persian rows to temp.csv for translation
    persian_rows[[text_column]].to_csv(temp_csv, index=False, encoding="utf-8-sig")

    # Translate Persian rows
    translated_csv = translate_csv(temp_csv, text_column=text_column, workers=workers)

    # Load translated text
    translated_df = pd.read_csv(translated_csv)

    # We only need the translated text under the same column name 'text'
```

```
    translated_text = translated_df[[translated_column]].rename(
        columns={translated_column: text_column}
    )

    # Combine English + translated rows (all under one column 'text')
    final_df = pd.concat(
        [english_rows[[text_column]], translated_text], ignore_index=True
    )

    # Save final CSV (only one column: 'text')
    final_df.to_csv(final_csv, index=False, encoding="utf-8-sig")

    # Clean up
    os.remove(temp_csv)
    os.remove(translated_csv)

    print(f"  Final CSV prepared (single 'text' column): {final_csv}")
    return final_csv
```

توضیح کد *prepare_final_csv*

۱. **تعریف تابع و پارامترها:**

```
def prepare_final_csv(
    input_csv: str, text_column="text", translated_column="translated", workers=2
):

```

- *input_csv*: مسیر فایل *CSV* ورودی.
- *text_column*: نام ستونی که متن‌ها در آن هستند (پیش‌فرض *"text"*).
- *translated_column*: نام ستونی که متن ترجمه‌شده بعداً در آن ذخیره می‌شود (پیش‌فرض *"translated"*).
- *workers*: تعداد رشته‌های موازی برای ترجمه (پیش‌فرض ۲).

۲. **بررسی وجود فایل:**

```
if not os.path.isfile(input_csv):
    raise FileNotFoundError(f"File '{input_csv}' does not exist.")

```

- اگر فایل ورودی وجود نداشته باشد، خطا می‌دهد و برنامه متوقف می‌شود.

۳. **خواندن فایل** *CSV*:

```
df = pd.read_csv(input_csv)

```

- فایل *CSV* را با *pandas* می‌خواند و آن را در *DataFrame* ذخیره می‌کند.

۴. **بررسی وجود ستون متن:**

```
if text_column not in df.columns:
    raise ValueError(
        f"Column '{text_column}' does not exist in the file. Columns:
    {df.columns.tolist()}"
    )

```

- بررسی می‌کند که آیا ستونی با نام *"text"* در فایل وجود دارد یا نه؛ اگر نه، خطا می‌دهد.

۵. **جدا کردن سطرهای فارسی و انگلیسی:**

```
english_rows = df[~df[text_column].apply(is_persian)]
persian_rows = df[df[text_column].apply(is_persian)]

```

- تابع *is_persian()* برای هر سطر بررسی می‌کند که آیا متن فارسی است یا نه.
- سطرهای انگلیسی در *english_rows* و سطرهای فارسی در *persian_rows* ذخیره می‌شوند.

۶. **تعریف مسیر فایل‌های خروجی:**

```
folder, filename = os.path.split(input_csv)
name, ext = os.path.splitext(filename)
final_csv = os.path.join(folder, f"{name}_final.csv")
temp_csv = os.path.join(folder, f"{name}_temp.csv")

```

- مسیر فایل ورودی را تجزیه می‌کند و مسیر فایل‌های موقت و نهایی را می‌سازد: **_temp.csv* (شامل فقط سطرهای فارسی برای ترجمه) و **_final.csv* (شامل خروجی نهایی انگلیسی).

۷. **ذخیره‌ی مستقیم سطرهای انگلیسی:**

```
english_rows[[text_column]].to_csv(final_csv, index=False, encoding="utf-8-sig")

```

- سطرهای انگلیسی بدون تغییر مستقیماً در فایل نهایی ذخیره می‌شوند.

۸. **در صورت نبود متن فارسی:**

```
if persian_rows.empty:
    print("No Persian text found. Final file saved.")
    return final_csv

```

- اگر در داده‌ها هیچ متن فارسی نباشد، مستقیماً فایل نهایی را برمی‌گرداند.

۹. **ذخیره‌ی سطرهای فارسی برای ترجمه:**

```
persian_rows[[text_column]].to_csv(temp_csv, index=False, encoding="utf-8-sig")

```

- تمام سطرهای فارسی را در یک فایل موقت (*temp_csv*) ذخیره می‌کند تا برای ترجمه استفاده شود.

۱۰. **ترجمه‌ی متون فارسی:**

```
translated_csv = translate_csv(temp_csv, text_column=text_column, workers=workers)

```

- فایل موقت فارسی را با تابع *translate_csv()* ترجمه می‌کند. این تابع به‌صورت موازی (*workers=2*) اجرا می‌شود و خروجی را به‌صورت *CSV* برمی‌گرداند.

۱۱. **خواندن خروجی ترجمه:**

```
translated_df = pd.read_csv(translated_csv)

```

- فایل ترجمه‌شده را می‌خواند و در *DataFrame* جدید ذخیره می‌کند.

۱۲. **تغییر نام ستون ترجمه به *"text"* :**

```
translated_text = translated_df[[translated_column]].rename(
    columns={translated_column: text_column}
)

```

- برای یک‌دست شدن، ستون *"translated"* را به *"text"* تغییر نام می‌دهد تا با سطرهای انگلیسی هماهنگ شود.

۱۳. **ترکیب داده‌های انگلیسی و ترجمه‌شده:**

```
final_df = pd.concat(
    [english_rows[[text_column]], translated_text], ignore_index=True
)

```

- هر دو مجموعه (انگلیسی و ترجمه‌شده) را زیر هم در یک *DataFrame* ادغام می‌کند.

۱۴. **ذخیره‌ی فایل نهایی:**

```
final_df.to_csv(final_csv, index=False, encoding="utf-8-sig")

```

- همه‌ی متن‌ها (الان همه انگلیسی هستند) را در یک فایل *CSV* نهایی ذخیره می‌کند.

۱۵. **حذف فایل‌های موقت:**

```
os.remove(temp_csv)
os.remove(translated_csv)

```

- فایل‌های موقتی (فارسی و ترجمه‌شده) را برای تمیزی محیط حذف می‌کند.

۱۶. **پیام نهایی و بازگرداندن مسیر:**

```
print(f" Final CSV prepared (single 'text' column): {final_csv}")
return final_csv

```

- آدرس فایل نهایی را چاپ و برمی‌گرداند.

تابع *is_persian* برای تشخیص متون فارسی این تابع با استفاده از عبارت منظم (*Regex*) بررسی می‌کند که آیا حداقل یک کاراکتر فارسی در متن وجود دارد یا خیر.

```
def is_persian(text: str) -> bool:
    """Return True if the text contains at least one Persian character."""
    if not text or not isinstance(text, str):
        return False
    return bool(re.search(r"[\u0600-\u06FF]", text))
```

منطق: این تابع با استفاده از عبارت منظم (*re.search(r"[\u0600-\u06FF]", text)*) بررسی می‌کند که آیا حداقل یک کاراکتر در محدوده‌ی یونیکد کاراکترهای فارسی و عربی (*0x0600* تا *0x06FF*) در متن وجود دارد یا خیر. در صورت مثبت بودن، مقدار *True* بازگردانده می‌شود.

- از ماژول *re* (عبارت منظم) استفاده می‌کند.
- الگوی *r"[\u0600-\u06FF]"*:
  ۱. محدوده‌ی یونیکد کاراکترهای فارسی و عربی (از *0x0600* تا *0x06FF*) را مشخص می‌کند.
  ۲. یعنی اگر حداقل یک کاراکتر در این محدوده وجود داشته باشد، تطبیق پیدا می‌کند.
- *re.search()*: اولین تطبیق را پیدا می‌کند و اگر باشد، شیء *match* برمی‌گرداند.
- *bool(...)*: تبدیل نتیجه به *True* یا *False*.

تابع *translate_csv* برای ترجمه‌ی موازی این تابع مسئول ترجمه‌ی متون یک ستون مشخص از فایل *CSV* به صورت موازی است و خروجی را در یک فایل جدید (*<original_name>_translated.csv*) ذخیره می‌کند.

```python
def translate_csv(input_csv: str, text_column="text", workers=4):
    """
    Takes a CSV file, translates the specified text column,
    and saves the output in the same folder as the input file with
    the name <original_name>_translated.csv.
    """
    if not os.path.isfile(input_csv):
        raise FileNotFoundError(f"File '{input_csv}' does not exist.")

    df = pd.read_csv(input_csv)

    if text_column not in df.columns:
        raise ValueError(
            f"Column '{text_column}' does not exist in the file. Columns:
    {df.columns.tolist()}"
        )

    texts = df[text_column].fillna("").astype(str).tolist()
    results = [""] * len(texts)

    # Use ThreadPoolExecutor to speed up translation (optional)
    if workers > 1:
        with ThreadPoolExecutor(max_workers=workers) as ex:
            futures = {
                ex.submit(translate_text, texts[i]): i for i in range(len(texts))
            }
            for fut in tqdm(
                as_completed(futures), total=len(futures), desc="Translating"
            ):
                idx = futures[fut]
                try:
                    results[idx] = fut.result()
                except Exception as e:
                    print(f" Error at index {idx}: {e}")
                    results[idx] = ""
    else:
        for i, t in enumerate(tqdm(texts, desc="Translating")):
            results[i] = translate_text(t)

    df["translated"] = results

    # Save output file in the same folder as the input file
    folder, filename = os.path.split(input_csv)
    name, ext = os.path.splitext(filename)
    output_csv = os.path.join(folder, f"{name}_translated{ext}")

    df.to_csv(output_csv, index=False, encoding="utf-8-sig")
    print(f" Translation complete. Saved to {output_csv}")
    return output_csv
```

توضیح: این تابع از *ThreadPoolExecutor* برای اجرای موازی تابع *translate_text* بر روی متون استفاده می‌کند تا سرعت فرآیند ترجمه افزایش یابد. نوار پیشرفت (*tqdm*) نیز برای نمایش وضعیت ترجمه استفاده می‌شود.

توضیح کد *translate_csv*

۱. **تعریف تابع و پارامترها:**

```python
def translate_csv(input_csv: str, text_column="text", workers=4):

```

- *input_csv*: مسیر فایل *CSV* ورودی.
- *text_column*: نام ستونی که متن‌ها در آن قرار دارند (پیش‌فرض *"text"*).
- *workers*: تعداد رشته‌های موازی (*threads*) برای ترجمه (پیش‌فرض ۴).

۲. **بررسی وجود فایل:**

```
if not os.path.isfile(input_csv):
    raise FileNotFoundError(f"File '{input_csv}' does not exist.")

```

- اگر فایل ورودی وجود نداشته باشد، خطا می‌دهد.

۳. **خواندن فایل *CSV*:**

```
df = pd.read_csv(input_csv)

```

- فایل *CSV* را با *pandas* خوانده و در یک *DataFrame* ذخیره می‌کند.

۴. **بررسی وجود ستون متن:**

```
if text_column not in df.columns:
    raise ValueError(
        f"Column '{text_column}' does not exist in the file. Columns:
    {df.columns.tolist()}"
    )

```

- مطمئن می‌شود ستونی که قرار است ترجمه شود در فایل وجود داشته باشد؛ در غیر این صورت خطا می‌دهد.

۵. **آماده‌سازی متن‌ها:**

```
texts = df[text_column].fillna("").astype(str).tolist()
results = [""] * len(texts)

```

- *texts*: تمام مقادیر ستون متن را به رشته تبدیل کرده و جای مقادیر خالی (*NaN*) را با رشته خالی پر می‌کند.
- *results*: آرایه‌ای برای ذخیره‌ی نتایج ترجمه، هم‌اندازه با تعداد سطرها.

۶. **ترجمه موازی (با *ThreadPoolExecutor*):**

```
if workers > 1:
    with ThreadPoolExecutor(max_workers=workers) as ex:
        futures = {
            ex.submit(translate_text, texts[i]): i for i in range(len(texts))
        }
        for fut in tqdm(
            as_completed(futures), total=len(futures), desc="Translating"
        ):
            idx = futures[fut]
            try:
                results[idx] = fut.result()
            except Exception as e:
                print(f" Error at index {idx}: {e}")
                results[idx] = ""

```

- اگر تعداد *workers* بیشتر از ۱ باشد، ترجمه به صورت موازی انجام می‌شود.
- هر متن به یک *thread* فرستاده می‌شود (*translate_text*).
- *tqdm* برای نمایش نوار پیشرفت استفاده شده است.
- در صورت بروز خطا، رشته خالی جایگزین می‌شود و پیام خطا چاپ می‌شود.

۷. **ترجمه تک‌نخی (بدون موازی):**

```
else:
    for i, t in enumerate(tqdm(texts, desc="Translating")):
        results[i] = translate_text(t)

```

- اگر *workers=1* باشد، ترجمه به صورت تک‌نخی انجام می‌شود، ولی باز هم نوار پیشرفت نشان داده می‌شود.

۸. **اضافه کردن ستون ترجمه:**

```
df["translated"] = results

```

- ستون جدید *"translated"* به *DataFrame* اضافه می‌شود و نتایج ترجمه در آن قرار می‌گیرند.

۹. **تعیین مسیر فایل خروجی:**

```
folder, filename = os.path.split(input_csv)
name, ext = os.path.splitext(filename)
output_csv = os.path.join(folder, f"{name}_translated{ext}")

```

- مسیر پوشه و نام فایل ورودی استخراج می‌شود.
- مسیر فایل خروجی به صورت *<original_name>_translated.csv* ساخته می‌شود.

۱۰. **ذخیره *CSV* ترجمه شده:**

```
df.to_csv(output_csv, index=False, encoding="utf-8-sig")
print(f" Translation complete. Saved to {output_csv}")

```

- *DataFrame* شامل متن اصلی و ترجمه در فایل جدید ذخیره می‌شود.
- پیام موفقیت چاپ می‌شود.

۱۱. **بازگرداندن مسیر فایل خروجی:**

```
return output_csv

```

- مسیر فایل *CSV* ترجمه‌شده را برمی‌گرداند.

تابع *translate_text* برای ترجمه‌ی یک آیتم متنی این تابع مسئول ترجمه‌ی یک رشته‌ی متنی با استفاده از زنجیره‌ی *vision_chain* و مدیریت خطاهای احتمالی از طریق تلاش مجدد (*retries*) است.

```
def translate_text(text: str, retries=2, pause=1.0) -> str:
    # ---------- Translate a single text ----------
    if not isinstance(text, str) or not text.strip():
        return ""

    prompt_input = {"text": text}

    attempt = 0
    while attempt <= retries:
        try:
            translated = vision_chain.invoke(prompt_input)
            return translated.strip()
        except Exception as e:
            attempt += 1
            print(f" Translation error (attempt {attempt}): {e}")
            if attempt > retries:
                return ""
            time.sleep(pause * attempt)
```

توضیح کد *translate_text*

۱. **تعریف تابع و پارامترها:**

```
def translate_text(text: str, retries=2, pause=1.0) -> str:

```

- *text*: رشته‌ای که قرار است ترجمه شود.
- *retries*: تعداد دفعات تلاش مجدد در صورت خطا (پیش‌فرض ۲).
- *pause*: مدت زمان مکث بین تلاش‌ها بر حسب ثانیه (پیش‌فرض ۱ ثانیه).
- **خروجی:** رشته‌ی ترجمه‌شده (*str*).

۲. **آماده‌سازی ورودی برای زنجیره ترجمه:**

```
prompt_input = {"text": text}
translated = vision_chain.invoke(prompt_input)
return translated.strip()

```

- متن ورودی در یک دیکشنری با کلید *"text"* قرار داده می‌شود.
- *vision_chain.invoke(prompt_input)*: متن را به زنجیره‌ی ترجمه می‌دهد و ترجمه نهایی را می‌گیرد.
- در صورت موفقیت، متن ترجمه‌شده را با *strip()*. برمی‌گرداند (حذف فاصله‌های اضافی).

۳. **مدیریت خطا و تلاش مجدد:**

```
except Exception as e:
    attempt += 1
    print(f" Translation error (attempt {attempt}): {e}")
    if attempt > retries:
        return ""
    time.sleep(pause * attempt)

```

- اگر خطایی رخ دهد (مانند قطع اتصال *API* یا خطای مدل):
  (آ) تعداد تلاش‌ها افزایش می‌یابد (*attempt += 1*).
  (ب) پیام خطا چاپ می‌شود.
  (ج) اگر تعداد تلاش‌ها بیشتر از مقدار *retries* باشد → رشته خالی برگردانده می‌شود.
  (د) قبل از تلاش مجدد، تابع به مدت *pause * attempt* ثانیه مکث می‌کند (استراتژی افزایش فاصله بین تلاش‌ها).

خلاصه‌ی مرحله اول حال مرحله ۱ به پایان می‌رسد و در پایان این مرحله ما دارای یک فایل *CSV* جدید هستیم که تمام داده‌های آن به انگلیسی ترجمه شده‌اند. این فایل در همان پوشه‌ای قرار دارد که فایل *CSV* اصلی قرار داشت.

مرحله دوم: تولید *Embedding* و ذخیره در *Weaviate* پس از اتمام مرحله اول، فایل *CSV* نهایی حاوی متون انگلیسی آماده شده است. خط زیر مسئول شروع مرحله دوم است:

```
process_texts(final_csv)
```

این دستور تابع *process_texts* را با مسیر فایل *CSV* نهایی (*final_csv*) فراخوانی می‌کند.

تابع *process_texts* این تابع مسئول اجرای موازی تولید *embedding* و ذخیره‌ی آیتم‌های متنی در دیتابیس است.

```
def process_texts(input_csv: str, max_workers: int = 8):
    """Process and store text embeddings in parallel."""
    df = pd.read_csv(input_csv)
    texts = df["text"].astype(str).tolist()

    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        futures = [
            executor.submit(store_text_item, f"text_{i+1}", text)
            for i, text in enumerate(texts)
        ]
```

```
        for _ in tqdm(
            as_completed(futures), total=len(futures), desc="  Processing texts"
        ):
            pass

    print(f"  Processed {len(texts)} text items.")
```

توضیح کد *process_texts*

۱. **تعریف تابع و پارامترها:**

```
def process_texts(input_csv: str, max_workers: int = 8):

```

- *input_csv*: مسیر فایل *CSV* ورودی که شامل ستون *"text"* است.
- *max_workers*: حداکثر تعداد رشته‌های موازی (*threads*) برای پردازش متون (پیش‌فرض ۸).

۲. **خواندن فایل *CSV* و آماده‌سازی متن‌ها:**

```
df = pd.read_csv(input_csv)
texts = df["text"].astype(str).tolist()

```

- فایل *CSV* ورودی را با *pandas* می‌خواند.
- ستون *"text"* را به رشته تبدیل و به لیست متن‌ها (*texts*) تبدیل می‌کند.

۳. **اجرای پردازش موازی:**

```
with ThreadPoolExecutor(max_workers=max_workers) as executor:
    futures = [
        executor.submit(store_text_item, f"text_{i+1}", text)
        for i, text in enumerate(texts)
    ]

```

- از *ThreadPoolExecutor* برای اجرای همزمان چند عملیات استفاده می‌شود.
- برای هر متن، تابع *store_text_item()* فراخوانی می‌شود.
- به هر متن یک شناسه یکتا (*text_1, text_2, ...*) داده می‌شود.
- هر فراخوانی به *executor.submit()* داده می‌شود تا در یک رشته‌ی مجزا اجرا شود.
- نتیجه‌ها در *futures* ذخیره می‌شوند (لیست *future objects*).

۴. **نمایش پیشرفت:**

```
for _ in tqdm(
    as_completed(futures), total=len(futures), desc="  Processing texts"
):
    pass

```

- حلقه بر روی *as_completed(futures)* می‌چرخد تا منتظر بماند تمام وظایف موازی تکمیل شوند.
- *tqdm* برای نمایش نوار پیشرفت استفاده شده است.

تابع *store_text_item* این تابع عملیات تولید *embedding* برای یک متن و ذخیره‌ی آن همراه با متادیتا در دیتابیس *Weaviate* را انجام می‌دهد.

```
def store_text_item(item_id: str, text_data: str):
    """Store text embedding and metadata in Weaviate."""
    embedding = get_embedding("text", text_data)
    properties = {
        "contentId": item_id,
        "modality": "text",
        "filePath": "",
        "content": text_data,
    }
    collection.data.insert(properties=properties, vector=embedding.tolist())
```

توضیح کد *store_text_item*

- **تعریف تابع و پارامترها:**
  - *item_id*: شناسه یکتا برای متن (مثلاً *"text_1"*).
  - *text_data*: خود متن مورد نظر برای پردازش.
- **تولید** *embedding*:
  - *embedding = get_embedding("text", text_data)*: بردار عددی (*embedding*) متن را تولید می‌کند.
  - این بردار، نمایش معنایی متن در فضای چندبعدی است و برای جستجوی معنایی و بازیابی متون استفاده می‌شود.
- **آماده‌سازی متادیتا:**
  - دیکشنری *properties* شامل اطلاعات جانبی است که همراه *embedding* ذخیره می‌شود: *contentId* (شناسه یکتا)، *modality* (نوع داده)، *filePath* (مسیر فایل منبع، که در اینجا خالی است) و *content* (خود متن).
- **ذخیره در** *Weaviate*:
  - *collection.data.insert()*: داده را در *collection* مشخص در *Weaviate* ذخیره می‌کند.
  - پارامتر *properties*: متادیتای متن.
  - پارامتر *vector*: بردار *embedding* متن (که با *tolist()*. به لیست تبدیل می‌شود تا قابل ذخیره باشد).

۵- تابع *get_embedding* این تابع مسئول تولید بردار *embedding* نرمالایز شده توسط مدل *CLIP* برای یک داده‌ی متنی است.

```python
def get_embedding(modality: str, input_data: Union[str, None]) -> np.ndarray:
    """Wrapper for modality-specific embedding."""
    mod = modality.lower()
    if mod == "text":
        """Return normalized CLIP text embedding."""
        if not isinstance(input_data, str):
            raise ValueError("`text` must be a string.")

        tokens = tokenizer([input_data]).to(DEVICE)
        with torch.no_grad():
            features = clip_model.encode_text(tokens)
            features = features / features.norm(dim=-1, keepdim=True)
        return features.detach().cpu().numpy().reshape(-1)
    else:
        raise ValueError("`modality` must be 'text'.")
```

توضیح کد *get_embedding*

- **تعریف تابع و پارامترها:**
  - *modality*: نوع داده.
  - *input_data*: داده ورودی برای تولید *embedding*
  - **خروجی:** آرایه‌ی *NumPy* (*np.ndarray*) که بردار عددی متن را باز می‌گرداند.
- **توکنایز کردن متن:**
  - *tokens = tokenizer([input_data]).to(DEVICE)*: متن را با *tokenizer* مدل *CLIP* به توکن تبدیل کرده و به دستگاه مناسب (*cuda* یا *cpu*) منتقل می‌کند.
- **تولید** *embedding* **با مدل** *CLIP*:
  - *with torch.no_grad()*: جلوگیری از ذخیره‌سازی گرادیان‌ها برای صرفه‌جویی در حافظه و زمان.
  - *features = clip_model.encode_text(tokens)*: تولید بردار عددی متن (*text embedding*) با مدل *CLIP*.
  - *features = features / features.norm(dim=-1, keepdim=True)*: نرمال‌سازی بردار (طول بردار برابر ۱ می‌شود).
- **بازگرداندن خروجی به** *NumPy*:
  - *features.detach().cpu().numpy().reshape(-1)*:

۱. *detach()*: جدا کردن از گراف محاسباتی *PyTorch*.
۲. *cpu()*: انتقال به حافظه *CPU*.
۳. *numpy()*: تبدیل به آرایه *NumPy*.
۴. *reshape(-1)*: تبدیل به بردار یک‌بُعدی.

۷ بخش بک اند اصلی

کد اصلی بک اند داخل پوشه server قرار دارد در زیربخش های بعدی به توضیح فایل‌های موجود رد پوشه server و عملکرد آنها پرداخته خواهدشد

۱.۷ فایل *config.py*

قبل از آغاز عملیات اجرایی، لازم است فایل *config.py* واقع در پوشه‌ی *server* بررسی و در صورت لزوم تنظیم گردد.
این فایل شامل مجموعه‌ای از تنظیمات اولیه و پیکربندی‌ها برای تعیین مسیر فایل‌ها و مشخص نمودن انواع مدل‌های مورد استفاده در پروژه است. در صورتی که روال‌های تعیین‌شده در بخش‌های پیشین پروژه به دقت دنبال شده باشد، نیازی به اعمال تغییرات در این مسیرها و تنظیمات نخواهد بود. در غیر این صورت، موارد درخواستی باید توسط کاربر تنظیم و به‌روزرسانی شوند.
کد زیر، محتوای کامل فایل *config.py* را نشان می‌دهد:

```python
import os
import torch
from dotenv import load_dotenv
from pathlib import Path

PROJECT_ROOT_PATH = Path.cwd().as_posix()
print(f"Project root path: {PROJECT_ROOT_PATH}")

WEAVIATE_COLLECTION_NAME = "Multimodal_Collection"

# === Load environment variables ===
load_dotenv() # Looks for .env in current working directory or parents

# === Device setup ===
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"

# === CLIP Model Config ===
CLIP_MODEL_NAME = "ViT-B-32"
PRETRAINED_LOCAL_PATH =
    f"{PROJECT_ROOT_PATH}/open_clip_weights/ViT-B-32-openai/open_clip_model.safetensors"

# === Whisper Model Config ===
WHISPER_MODEL = "small"

# === LLM Config ===
LLM_MODEL_NAME = "openai/gpt-5-image-mini"
LLM_API_BASE = "https://openrouter.ai/api/v1"
LLM_API_KEY = os.getenv("LLM_API_KEY") # <- Loaded from .env

if not LLM_API_KEY:
  raise ValueError(" Missing LLM_API_KEY in .env file!")

print(" Environment key loaded successfully!")
```

این قطعه کد به منظور پیکربندی (*Configuration*) یک پروژه‌ی چندحالته (*Multimodal*) طراحی شده است که بر مبنای مدل‌های زبان بزرگ (*LLM*) و مدل‌های پردازش تصویر (مانند *Multimodal RAG* یا چت‌بات چندحالته) عمل می‌کند.

۱.۱.۷ مسیر ریشه پروژه

```python
PROJECT_ROOT_PATH = Path.cwd().as_posix()
print(f"Project root path: {PROJECT_ROOT_PATH}")
```

- *Path.cwd()*: مسیر جاری عملیاتی (*Current Working Directory*) را بازیابی می‌نماید.
- *as_posix()*.: مسیر را به فرم استاندارد یونیکس (/) تبدیل می‌کند تا سازگاری آن در سیستم‌عامل‌های مختلف (ویندوز/لینوکس) تضمین گردد.

۲.۱.۷ نام کالکشن پایگاه داده *Weaviate*

```
WEAVIATE_COLLECTION_NAME = "Multimodal_Collection"
```

این نام برای کالکشنی است که داده‌های چندحالته (شامل متن، تصویر، صدا) در پایگاه داده *Weaviate* در آن ذخیره می‌شوند و متناظر با همان *collection* ایجاد شده در مراحل پیشین است.

۳.۱.۷ بارگذاری متغیرهای محیطی از *env*.

```
load_dotenv()
```

این تابع به دنبال فایلی به نام *env*. در پوشه‌ی جاری یا والد آن جستجو کرده و تمام متغیرهای محیطی تعریف‌شده در آن را وارد محیط اجرایی می‌کند. فایل مذکور در مراحل اولیه پروژه ایجاد و کلید *API* لازم در آن تنظیم شده است.

۴.۱.۷ تنظیم دستگاه اجرا (*GPU* یا *CPU*)

```
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
```

- اگر سیستم مجهز به واحد پردازش گرافیکی (*GPU*) با پشتیبانی از *CUDA* باشد، از *GPU* استفاده خواهد شد.
- در غیر این صورت، عملیات بر روی واحد پردازش مرکزی (*CPU*) انجام می‌پذیرد.

۵.۱.۷ پیکربندی مدل *CLIP*

```
CLIP_MODEL_NAME = "ViT-B-32"
PRETRAINED_LOCAL_PATH =
    f"{PROJECT_ROOT_PATH}/open_clip_weights/ViT-B-32-openai/open_clip_model.safetensors"
```

- *CLIP_MODEL_NAME*: مدل *CLIP* منتخب که در اینجا *Vision Transformer* با اندازه‌ی *Base* و اندازه‌ی پچ ۳۲ (*ViT-B-32*) است.
- *PRETRAINED_LOCAL_PATH*: مسیر فایل وزن‌های از پیش‌آموزش‌دیده مدل است که به صورت محلی در پوشه‌ی پروژه ذخیره شده است. فایل *open_clip_model.safetensors* حاوی وزن‌های از پیش‌آموزش‌دیده مدل است.
- مدل *CLIP* به منظور تبدیل تصویر و متن به یک فضای برداری مشترک استفاده می‌گردد.

۶.۱.۷ پیکربندی مدل *Whisper*

```
WHISPER_MODEL = "small"
```

- این مدل از *OpenAI* برای تبدیل گفتار به متن (*Speech-to-Text*) به کار می‌رود.
- مقدار *WHISPER_MODEL* می‌تواند یکی از مقادیر *small*، *medium* یا *large* باشد.
- مدل‌های *medium* و *large* برای بارگذاری روی *CPU* مناسب نبوده و به *GPU* نیاز دارند.
- مدل *small* برای بارگذاری روی *CPU* مناسب است اما دقت کمتری نسبت به دو مدل دیگر دارد. این مدل برای داده‌های صوتی به زبان انگلیسی عملکرد مطلوبی دارد، اما برای زبان‌هایی غیر از انگلیسی (مانند فارسی) دارای عملکرد ضعیف‌تری است.

۷.۱.۷ پیکربندی مدل زبانی بزرگ (*LLM*)

```
LLM_MODEL_NAME = "openai/gpt-5-image-mini"
LLM_API_BASE = "https://openrouter.ai/api/v1"
LLM_API_KEY = os.getenv("LLM_API_KEY")
```

- *LLM_MODEL_NAME*: مدل زبانی مورد استفاده است که در اینجا نسخه‌ی کوچک از *GPT-5* با قابلیت پشتیبانی از تصویر می‌باشد.
- *LLM_API_BASE*: آدرس سرور *OpenRouter* برای فراخوانی *API* مدل‌ها.
- *LLM_API_KEY*: از فایل *env*. خوانده می‌شود و برای احراز هویت *API* الزامی است.

۲.۷ فایل *ai_models.py*

در این پروژه، مدل *CLIP* برای تولید بردارهای جاسازی (*embedding*) و مدل *Whisper* برای تبدیل صوت به متن استفاده شده است.

این دو مدل با استفاده از کد زیر بارگذاری شده‌اند. این کد در فایل *ai_models.py* واقع در پوشه‌ی *server* قرار دارد.

```
import whisper
import open_clip
from .config import WHISPER_MODEL, CLIP_MODEL_NAME, PRETRAINED_LOCAL_PATH, DEVICE

""" Whisper """
whisper_model = whisper.load_model(WHISPER_MODEL, device=DEVICE)

""" Open_CLIP """
clip_model, _, preprocess = open_clip.create_model_and_transforms(
  CLIP_MODEL_NAME, pretrained=PRETRAINED_LOCAL_PATH
)
tokenizer = open_clip.get_tokenizer(CLIP_MODEL_NAME)
clip_model.to(DEVICE)
clip_model.eval()
```

این قطعه کد وظیفه‌ی بارگذاری و آماده‌سازی مدل‌های چندحالته (صوت و تصویر) را برعهده دارد که در یک سیستم مبتنی بر بازیابی پیشرفته‌ی چندحالته (مانند *Multimodal RAG Chatbot*) مورد استفاده قرار می‌گیرد. ساختار این فایل به‌گونه‌ای طراحی شده که مدل‌ها تنها یک‌بار بارگذاری شوند و در طول چرخه‌ی عمر پروژه آماده‌ی استفاده باشند.

۱.۲.۷ بارگذاری مدل *Whisper* (تبدیل گفتار به متن)

```
""" Whisper """
whisper_model = whisper.load_model(WHISPER_MODEL, device=DEVICE)
```

- *whisper.load_model()*: مدل انتخاب‌شده (*WHISPER_MODEL*) را بارگذاری می‌کند (مانند *small*، *base*، *medium*، *large*).
- پارامتر *device=DEVICE* تضمین می‌کند که مدل بر روی واحد پردازش گرافیکی (*GPU*) یا مرکزی (*CPU*) مناسب تنظیم شود.
- خروجی *whisper_model* یک شیء مدل است که قابلیت تبدیل ورودی صوتی به متن را فراهم می‌آورد.

۲.۲.۷ بارگذاری مدل *CLIP* (درک تصویر و متن)

```
""" Open_CLIP """
clip_model, _, preprocess = open_clip.create_model_and_transforms(
    CLIP_MODEL_NAME, pretrained=PRETRAINED_LOCAL_PATH
)
```

- *open_clip.create_model_and_transforms()* سه مقدار را برمی‌گرداند:
  ۱. مدل *CLIP*: قابلیت تبدیل متن و تصویر به بردارهای مشترک در یک فضای برداری یکسان را دارد.
  ۲. مدیریت کلاس‌ها (*head*): که در اینجا با استفاده از _ نادیده گرفته شده است زیرا مورد استفاده‌ی مستقیم قرار نمی‌گیرد.
  ۳. تابع *preprocess*: برای پیش‌پردازش تصویر (*resize*، *normalization* و غیره) پیش از ورود به مدل استفاده می‌شود.
- پارامتر *pretrained=PRETRAINED_LOCAL_PATH* باعث می‌شود وزن‌های مدل از فایل محلی (*safetensors*.) بارگذاری شوند.

۳.۲.۷ بارگذاری *Tokenizer* مدل *CLIP*

```
tokenizer = open_clip.get_tokenizer(CLIP_MODEL_NAME)
```

- توکنایزر (*Tokenizer*) مسئول تبدیل متن به ورودی عددی (*tokens*) قابل فهم برای مدل *CLIP* است.
- به عنوان مثال، برای توکن‌سازی یک متن: *text = tokenizer(["a dog running in the park"])*.
- در مدل *CLIP*، متن و تصویر هر دو به یک فضای برداری مشترک نگاشت می‌شوند تا امکان اندازه‌گیری میزان شباهت معنایی بین آن‌ها فراهم گردد.

۴.۲.۷ انتقال مدل به *GPU* و حالت *evaluation*

```
clip_model.to(DEVICE)
clip_model.eval()
```

- *to(DEVICE)*.: مدل را مطابق با پیکربندی موجود در فایل *config* به *GPU* یا *CPU* منتقل می‌کند.
- *eval()*.: مدل را در حالت استنتاج (*inference*) قرار می‌دهد، که منجر به:
  - غیرفعال شدن مکانیسم‌هایی نظیر *Dropout* و *BatchNorm* می‌شود.
  - آماده‌سازی مدل صرفاً برای پیش‌بینی و نه برای فرآیند آموزش.

۳.۷ فایل *database.py*

برای بازیابی (*retrieve*) فایل‌هایی که در مراحل پیشین به پایگاه داده (*database*) وارد شده‌اند، لازم است که به نمونه‌ی ایجاد شده از پایگاه داده متصل شویم.
کد زیر، که در فایل *database.py* در پوشه‌ی *server* قرار دارد، همین وظیفه را بر عهده دارد:

```
try:
  weaviate_client = weaviate.connect_to_local()
  print(" Connected to Weaviate")
except Exception as e:
  print(f" Weaviate connection failed: {e}")
  weaviate_client = None
```

۱.۳.۷ توضیح دقیق عملکرد

اتصال به *Weaviate* محلی:

```
weaviate_client = weaviate.connect_to_local()
```

- این تابع تلاش می‌کند به نسخه‌ی محلی (*local*) از پایگاه داده‌ی وکتوری *Weaviate* که بر روی سیستم یا از طریق *Docker* در حال اجرا است، متصل شود.
- به‌صورت پیش‌فرض، آدرس اتصال *http://localhost:8080* در نظر گرفته شده است.
- در صورت موفقیت‌آمیز بودن اتصال، یک شیء کلاینت (*client object*) برگردانده می‌شود که امکان تعامل با پایگاه داده را فراهم می‌سازد.

۴.۷ فایل *main.py*

پیش از اجرای پروژه، لازم است اطمینان حاصل شود که داکر (*Docker*) بر روی سیستم شما در حال اجرا است، زیرا اتصال به پایگاه داده (*database*) مستلزم فعال بودن داکر است.
برای اجرای پروژه، یک واسط خط فرمان (*cmd*) باز کنید، به پوشه‌ی ریشه‌ی پروژه بروید و دستور زیر را اجرا نمایید:

```
python -m server.main
```

سپس، یک مرورگر وب باز کرده و به آدرس زیر مراجعه نمایید:

```
localhost:8000
```

با اجرای دستور *python -m server.main*، فایل اصلی پروژه یعنی *main.py* که در پوشه‌ی *server* قرار دارد، اجرا می‌شود.

۱.۴.۷ کد فایل *main.py*

کد زیر، محتوای کامل فایل *main.py* را نشان می‌دهد:

```
import os
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from fastapi.responses import HTMLResponse
from fastapi.staticfiles import StaticFiles
from .search_routes import router as search_router

```

```
app = FastAPI(title="Multimodal RAG Chatbot")

# CORS
app.add_middleware(
  CORSMiddleware,
  allow_origins=["*"],
  allow_credentials=True,
  allow_methods=["*"],
  allow_headers=["*"],
)

# Static mounts
app.mount("/client", StaticFiles(directory="client"), name="client")
app.mount("/content", StaticFiles(directory="content"), name="content")

# Routes
app.include_router(search_router)

@app.get("/", response_class=HTMLResponse)
async def home():
  # Serve the index.html file properly
  html_path = os.path.join("client", "index.html")
  if os.path.exists(html_path):
    with open(html_path, encoding="utf-8") as f:
      return HTMLResponse(f.read())
  else:
    return HTMLResponse("<h1>Frontend not found</h1>", status_code=404)

if __name__ == "__main__":
  import uvicorn
  uvicorn.run(app, host="0.0.0.0", port=8000)
```

۲.۴.۷ توضیح کد

ساخت اپلیکیشن *FastAPI*:

```
app = FastAPI(title="Multimodal RAG Chatbot")
```

یک شیء *FastAPI* با عنوان مشخص‌شده ساخته می‌شود. این شیء نمایانگر کل برنامه‌ی وب (بک اِند) پروژه است.

تنظیم *CORS*:

```
app.add_middleware(
  CORSMiddleware,
  allow_origins=["*"],
  allow_credentials=True,
  allow_methods=["*"],
  allow_headers=["*"],
)
```

این بخش مجوزهای لازم را برای دسترسی همه‌ی درخواست‌ها (*from any domain*) به *API* فراهم می‌کند. جزئیات این تنظیمات به شرح زیر است:

- *allow_origins=["*"]*: تمامی دامنه‌ها مجاز به ارسال درخواست هستند.
- *allow_credentials=True*: ارسال کوکی‌ها در درخواست‌ها مجاز است.
- *allow_methods=["*"]*: تمامی متدهای *HTTP* (شامل *GET*، *POST*، و غیره) مجازند.
- *allow_headers=["*"]*: تمامی سرآیندهای (*headers*) ارسالی مجاز هستند.

سرو فایل‌های استاتیک

```
app.mount("/client", StaticFiles(directory="client"), name="client")
app.mount("/content", StaticFiles(directory="content"), name="content")
```

با استفاده از این دو خط، دو مسیر مجزا برای فایل‌های استاتیک تعریف شده است:

- محتوای پوشه‌ی *client* در مسیر */client* قابل دسترسی است.
- محتوای پوشه‌ی *content* در مسیر */content* قابل دسترسی است.

افزودن مسیرهای دیگر از فایل جداگانه

```
app.include_router(search_router)
```

این دستور کلیه‌ی مسیرهای تعریف‌شده در فایل *search_routes.py* را به ساختار اصلی اپلیکیشن اضافه می‌کند.

تعریف مسیر اصلی (/)

```
@app.get("/", response_class=HTMLResponse)
async def home():
  html_path = os.path.join("client", "index.html")
  if os.path.exists(html_path):
    with open(html_path, encoding="utf-8") as f:
      return HTMLResponse(f.read())
  else:
    return HTMLResponse("<h1>Frontend not found</h1>", status_code=404)
```

در هنگام مراجعه‌ی کاربر به مسیر ریشه (/)، عملکرد به صورت زیر است:

- برنامه به دنبال فایل *client/index.html* می‌گردد.
- اگر فایل مورد نظر یافت شود، محتوای آن به عنوان پاسخ *HTML* بازگردانده و نمایش داده می‌شود.
- در غیر این صورت، پیام خطای *Frontend not found* به همراه کد وضعیت *404* بازگردانده خواهد شد.

اجرای برنامه با *Uvicorn*

```
if __name__ == "__main__":
  import uvicorn
  uvicorn.run(app, host="0.0.0.0", port=8000)
```

این بلوک تضمین می‌کند که در صورت اجرای مستقیم فایل (*python main.py*):

- سرور *FastAPI* توسط *Uvicorn* اجرا گردد.
- سرور بر روی تمامی *IP*های محلی (*0.0.0.0*) و پورت *8000* در دسترس باشد.

۳.۴.۷ قابلیت‌های جستجو و درخواست‌ها

پس از دسترسی به آدرس *localhost:8000* در مرورگر، می‌توان درخواست‌هایی با ترکیب‌های چندحالته (*multimodal*) زیر را به سیستم ارسال کرد. درخواست‌ها می‌توانند شامل هر ترکیبی از متن، صوت و تصویر باشند:

۱. فقط متن

۲. فقط تصویر

۳. فقط فایل صوتی

۴. متن + تصویر

۵. متن + فایل صوتی

۶. تصویر + فایل صوتی

۷. متن + تصویر + فایل صوتی

توجه: ورود تعداد زیادی فایل صوتی یا تصویری به طور همزمان توصیه نمی‌شود، هرچند محدودیتی در تعداد وجود ندارد. دلایل عدم توصیه به شرح زیر است:

- این اقدام می‌تواند منجر به کاهش کارایی پاسخ‌دهی سیستم (*response*) گردد (که جزئیات آن در ادامه بررسی خواهد شد).
- تمام داده‌های تصویری به مدل زبان بزرگ (*LLM*) ارسال می‌شوند. در صورت زیاد بودن حجم داده‌ها، احتمال اتمام توکن‌های مجاز برای استفاده‌ی رایگان از مدل وجود دارد و *LLM* برای مدتی پاسخگو نخواهد بود.

۵.۷ فایل *search_routes.py* و *utils.py*

هنگامی که یک درخواست جستجو در بخش فرانت (*Frontend*) ایجاد می‌شود، از طریق جاوا اسکریپت (*JavaScript*) به مسیر *API* زیر درخواست زده می‌شود:

```
@router.post("/multimodal")
async def search_multimodal(
    query: str = Form(default=""), files: List[UploadFile] = File(default=[])
):
```

این تابع در فایل *search_routes.py* واقع در پوشه‌ی *server* تعریف شده است.

۱.۵.۷ توضیح کلی عملکرد مسیر */multimodal*

این مسیر یک درخواست چندمودال (*multimodal*) را دریافت می‌کند (شامل متن و/یا فایل‌های آپلودشده مانند تصویر/صوت). ابتدا ورودی‌ها پردازش شده و به بردارهای جاسازی (*embedding*) تبدیل می‌شوند. سپس یک بردار نهایی (تک یا میانگین بردارها) ساخته شده، با آن در پایگاه داده‌ی وکتوری (*Vector Database*) جستجو انجام می‌شود. در نهایت، نتایج نرمال‌سازی شده، داده‌های لازم برای مدل زبان بزرگ (*LLM*) آماده و به آن فرستاده می‌شود و پاسخ *JSON* شامل نتایج بازیابی شده و خروجی *LLM* به کلاینت بازگردانده می‌شود. در انتهای فرآیند، فایل‌های موقتی حذف می‌گردند.

۲.۵.۷ بررسی گام‌به‌گام کد

۱. امضای تابع و ورودی‌ها

```
@router.post("/multimodal")
async def search_multimodal(
    query: str = Form(default=""),
    files: List[UploadFile] = File(default=[])
):
```

- این یک مسیر *POST* است که یک فیلد فرم *query* (متن آزاد) و لیستی از فایل‌های آپلودشده (*files*) را دریافت می‌کند.
- *UploadFile* یک کلاس استاندارد از *FastAPI* است که برای مدیریت فایل‌های آپلودی استفاده می‌شود.
- تابع به صورت ناهمگام (*async*) تعریف شده است تا امکان استفاده از عملیات‌های غیرهمزمان (مانند *await save_temp_file(file)*) فراهم گردد.

۲. آماده‌سازی متغیرها

- *temp_paths*: مسیر فایل‌های موقتی که برای پاکسازی نهایی روی دیسک ذخیره شده‌اند.
- *embeddings*: لیستی از بردارهای embedding استخراج‌شده از متن، تصویر یا صوت.
- *response_data*: دیکشنری خروجی که در نهایت به فرانت بازگردانده می‌شود.

۳. تشخیص وجود متن و پردازش آن اگر درخواست ورودی حاوی متن باشد، بخش زیر اجرا می‌شود:

```
has_text = bool(query and query.strip())

if has_text:
    english_query = normalize_text_to_english(query)
    text_emb = embed_text(english_query)
    embeddings.append(text_emb)
    response_data["original_query"] = query
    response_data["processed_query"] = english_query
```

- از تابع *normalize_text_to_english* برای ترجمه‌ی متن استفاده می‌شود. در نتیجه اگر متن ورودی به زبانی غیر انگلیسی باشد، ابتدا به انگلیسی ترجمه می‌گردد.
- سپس برای متن انگلیسی‌شده، با استفاده از متد *embed_text* یک بردار جاسازی (*embedding*) به دست می‌آید.
- *embedding* متن به لیست *embeddings* اضافه می‌شود و اطلاعات اصلی و پردازش‌شده در *response_data* ذخیره می‌گردد.

۴. پردازش فایل‌های آپلودی (ذخیره موقت و استخراج *embedding*)

```
for file in files:
    temp_path = await _save_temp_file(file)
    temp_paths.append(temp_path)

    if file.content_type.startswith("image/"):
        img_emb = embed_image(temp_path)
        embeddings.append(img_emb)
        try:
            img_obj = Image.open(temp_path).copy()
        except Exception as e:
            img_obj = None
        image_files.append(img_obj)

    elif file.content_type.startswith("audio/"):
        audio_emb, audio_meta = _process_audio(temp_path)
        embeddings.append(audio_emb)
        audio_transcriptions.append(audio_meta)
```

- هر فایل ابتدا با *save_temp_file* به صورت موقت ذخیره شده و مسیر آن برای پاکسازی نهایی نگهداری می‌شود.
- **برای تصاویر:** تابع *embed_image* بردار تصویر را ساخته و به لیست اضافه می‌کند. همچنین، تصویر با کتابخانه‌ی *PIL* باز شده و در *image_files* نگهداری می‌شود تا برای ارسال به *LLM* یا پیش‌نمایش در دسترس باشد.
- **برای صدا:** ابتدا با استفاده از تابع *process_audio*، بردار جاسازی و اطلاعات فرادادهای (*metadata*) استخراج و به لیست‌های مربوطه اضافه می‌شوند. این فرادادہ حاوی نام فایل، متن فایل به زبان اصلی، متن فایل به زبان انگلیسی و زبان شناسایی‌شده‌ی فایل است که در ادامه توضیح داده خواهد شد.

۵. به دست آوردن متن فایل‌های صوتی

- اگر تنها یک فایل صوتی وجود داشته و هیچ متن ورودی‌ای توسط کاربر ارائه نشده باشد، مستقیماً نتیجه‌ی transcript (*transcription*) در *response_data* قرار می‌گیرد.
- در غیر این صورت، تمامی transcription ها در قالب یک لیست بازگردانده می‌شوند.

۶. تعیین بردار نهایی جستجو (*single vs multimodal mean*)

```
if not embeddings:
    return JSONResponse({"error": "No valid text or media found to process."},
    status_code=400)

if len(embeddings) == 1:
    q_emb = embeddings[0]
    response_data["search_mode"] = "single"
else:
    q_emb_array = np.array(embeddings)
    mean_emb = np.mean(q_emb_array, axis=0)
    q_emb = mean_emb
    response_data["search_mode"] = f"multimodal_mean ({len(embeddings)} inputs)"
```

- اگر هیچ بردار جاسازی (*embedding*) تولید نشده باشد، خطای *400* بازگردانده می‌شود.
- اگر تنها یک بردار وجود داشته باشد، همان به عنوان بردار جستجوی نهایی (*q_emb*) تعیین می‌شود (*single mode*).
- اگر چندین بردار وجود داشته باشد، میانگین این بردارها گرفته می‌شود تا یک بردار واحد چندمودال ایجاد گردد.
- **نکته:** این رویکرد (میانگین‌گیری) ساده‌ترین روش ترکیب مودال‌ها است. مشکل آن این است که اگر ورودی‌ها دارای مفاهیم دور از هم باشند، بردار میانگین ممکن است دقت بازیابی مناسبی نداشته باشد. این بردار نهایی (*q_emb*) برای جستجو در پایگاه داده استفاده خواهد شد.

۷. اجرای جستجو در پایگاه داده برداری با استفاده از بردار نهایی *q_emb*، جستجو برای هر مودال به صورت جداگانه اجرا می‌شود:

```
text_res = search_by_embedding(q_emb, "text", top_k=5)
image_res = search_by_embedding(q_emb, "image", top_k=5)
audio_res = search_by_embedding(q_emb, "audio", top_k=5)
```

- تابع *search_by_embedding* فراخوانی می‌شود. این تابع بردار نهایی را دریافت کرده و بر اساس مدالیته مشخص‌شده (*text*، *image*، *audio*)، تعداد 5 داده‌ی مشابه را از پایگاه داده بازیابی می‌کند.
- این فرآیند تضمین می‌کند که از هر نوع مدالیته، ۵ داده‌ی نزدیک به ورودی کاربر بازیابی گردد.

۸. نرمال‌سازی نتایج و آماده‌سازی برای *LLM* سپس با استفاده از تابع *normalize_results*، نتایج به فرمت یکپارچه برای ارسال به *LLM* و فرانت تبدیل و در *response_data* ذخیره می‌شوند.

```
normalized_texts = _normalize_results(text_res)
normalized_images = _normalize_results(image_res)
normalized_audios = _normalize_results(audio_res)

response_data.update({
    "text_results": normalized_texts,
    "image_results": normalized_images,
    "audio_results": normalized_audios,
})
```

۹. فیلتر کردن و آماده‌سازی داده‌های نهایی برای *LLM*
در این مرحله، ما دارای موارد زیر هستیم:

۱. **تمام کوئری‌های کاربر** (متن، تصویر، صوت): کوئری متن در متغیر *query*، کوئری‌های تصویر در لیست *image_files* و متن تبدیل‌شده‌ی صوت در لیست *audio_transcriptions* ذخیره شده‌اند.

۲. **یک لیست ۵ تایی از متون بازیابی‌شده** (*normalized_texts*) که از دیتابیس بازیابی و نرمال شده‌اند.

۳. **یک لیست ۵ تایی از تصاویر بازیابی‌شده** (*normalized_images*) که از دیتابیس بازیابی و نرمال شده‌اند.

۴. **یک لیست ۵ تایی از متون مربوط به صوت‌های بازیابی‌شده** (*normalized_audios*) که از دیتابیس بازیابی و نرمال شده‌اند.

برای جلوگیری از ارسال حجم زیاد داده به *LLM* و مدیریت توکن‌ها، تنها زیرمجموعه‌ای از داده‌ها (کوئری‌های کاربر و فقط یک مورد از هر نوع داده بازیابی شده) در دیکشنری *llm_data* ذخیره و به *LLM* ارسال می‌شوند.
داده‌هایی که در دیکشنری *llm_data* ذخیره شده و به *LLM* فرستاده می‌شوند عبارت‌اند از:

- تمام داده‌های کاربر.
- یک مورد از متون بازیابی‌شده (*normalized_texts[0]*).
- یک مورد از تصاویر بازیابی‌شده (*normalized_images[0]*).
- یک مورد از متون صوتی بازیابی‌شده (*normalized_audios[0]*).

همچنین، یک نسخه دیگر از این داده‌ها به نام *llm_data_modified_for_front* ساخته می‌شود که همان محتویات داده‌های بازیابی‌شده را دارد اما **بدون کوئری‌های کاربر**، که صرفاً برای راحتی کار در فرانت (*Frontend*) ایجاد شده است.

```
llm_data = {
    "user_text_query": query if has_text else None,
    "user_image_queries": image_files if image_files else None,
    "user_audio_queries": (
        [a.get("processed_text") for a in audio_transcriptions]
        if audio_transcriptions
        else None
    ),
    "retrieved_texts": (
        [t.get("content") for t in normalized_texts[:1]]
        if normalized_texts
        else None
    ),
    "retrieved_images": (
        [
            (
                Image.open(PROJECT_ROOT_PATH + i.get("filePath"))
                if i.get("filePath")
                else (
                    Image.open(requests.get(i.get("url"), stream=True).raw)
                    if i.get("url")
                    else None
                )
            )
```

```
            for i in normalized_images[:1]
        ]
        if normalized_images
        else None
    ),
    "retrieved_audios": (
        [
            a.get("content") or a.get("processed_text")
            for a in normalized_audios[:1]
        ]
        if normalized_audios
        else None
    ),
}

llm_data_modified_for_front = {
    "retrieved_texts": (
        [t.get("content") for t in normalized_texts[:1]]
        if normalized_texts
        else None
    ),
    "retrieved_images": (
        [i.get("filePath") or i.get("url") for i in normalized_images[:1]]
        if normalized_images
        else None
    ),
    "retrieved_audios": (
        [a.get("filePath") for a in normalized_audios[:1]]
        if normalized_audios
        else None
    ),
}
```

۱۰. **ارسال داده به *LLM* و بازگرداندن پاسخ** دیکشنری *llm_data* به تابع *feed_data_into_llm* ارسال می‌شود. خروجی این تابع که پاسخ متنی *LLM* است، در *response_data["llm_response"]* برای نمایش در فرانت ذخیره می‌شود.

```
response_data["llm_response"] = feed_data_into_llm(llm_data)
```

۳.۵.۷ توابع کمکی (*Utility Functions*) در *utils.py*

در فرآیند بالا، از توابع کمکی متعددی استفاده شده است که در ادامه توضیح داده می‌شوند:

۱. **تابع** *normalize_text_to_english* هدف: نرمال‌سازی متن ورودی به زبان انگلیسی از طریق ترجمه.

```
def normalize_text_to_english(text: str) -> str:
  """
  Convert any input text into English if it is not already English.
  Works offline for language detection + uses Google Translate API.
  """
  if not text or text.strip() == "":
    return text

  try:
    translated = GoogleTranslator(source='auto', target='en').translate(text)
    return translated
  except Exception as e:
    print(f"Translation error: {e}")
    return text
```

- این تابع از *GoogleTranslator* (از پکیج *deep-translator*) استفاده می‌کند.
- *source='auto'* زبان ورودی را به صورت خودکار تشخیص می‌دهد.
- *target='en'* تضمین می‌کند که خروجی همیشه به زبان انگلیسی باشد.

۲. **تابع** *embed_text* هدف: تبدیل یک رشته متنی به یک بردار عددی (*embedding*) با استفاده از مدل *CLIP*.

```
def _to_numpy(t: torch.Tensor) -> np.ndarray:
  return t.detach().cpu().numpy().reshape(-1)

def embed_text(text: str) -> np.ndarray:
```

```
  tokens = tokenizer([text]).to(DEVICE)
  with torch.no_grad():
    text_features = clip_model.encode_text(tokens)
    text_features /= text_features.norm(dim=-1, keepdim=True)
  return _to_numpy(text_features)
```

- *tokens = tokenizer([text]).to(DEVICE)*: متن را به *tokens* تبدیل کرده و به دستگاه مناسب (*CPU/GPU*) منتقل می‌کند.
- *with torch.no_grad()*: برای صرفه‌جویی در منابع و سرعت، محاسبه‌ی گرادیان‌ها غیرفعال می‌شود.
- *text_features = clip_model.encode_text(tokens)*: ویژگی‌های متنی توسط مدل *CLIP* استخراج می‌شوند.
- نرمال‌سازی بردارها: طول (*norm*) هر بردار به *1* نرمال می‌شود تا برای مقایسه‌ی شباهت کسینوسی (*cosine similarity*) مناسب باشد.
- *return _to_numpy(text_features)*: تنسور *PyTorch* به آرایه‌ی *NumPy* تبدیل می‌گردد.

۳. **تابع** *embed_image* هدف: تبدیل یک تصویر به یک بردار عددی (*embedding*) با استفاده از مدل *CLIP*.

```
def embed_image(image_path: str) -> np.ndarray:
  img = Image.open(image_path).convert("RGB")
  x = preprocess(img).unsqueeze(0).to(DEVICE)
  with torch.no_grad():
    image_features = clip_model.encode_image(x)
    image_features /= image_features.norm(dim=-1, keepdim=True)
  return _to_numpy(image_features)
```

- *img = Image.open(image_path).convert("RGB")*: تصویر با *Pillow* باز و به فرمت *RGB* تبدیل می‌شود.
- *x = preprocess(img).unsqueeze(0).to(DEVICE)*: تصویر توسط تابع *preprocess* (دریافت شده از *open_clip*) پیش‌پردازش می‌شود و به دستگاه منتقل می‌گردد. *unsqueeze(0)*. بعد *batch* را اضافه می‌کند.
- *image_features = clip_model.encode_image(x)*: ویژگی‌های تصویری توسط مدل *CLIP* استخراج می‌شوند.
- بردار ویژگی‌ها نرمال‌سازی شده و در نهایت به آرایه‌ی *NumPy* تبدیل می‌شود.

۴. **تابع** *save_temp_file* هدف: ذخیره‌ی فایل آپلودشده (*UploadFile*) در یک مسیر موقت و بازگرداندن مسیر آن.

```
async def save_temp_file(file: UploadFile) -> str:
  """Saves UploadFile to a temp path and returns the path."""
  # Ensure a file extension, default to .tmp if none
  suffix = os.path.splitext(file.filename)[1] or ".tmp"
  with tempfile.NamedTemporaryFile(delete=False, suffix=suffix) as tf:
    tf.write(await file.read())
    return tf.name
```

- تابع *asynchronous* است و از *await* برای خواندن محتوای فایل استفاده می‌کند.
- با *tempfile.NamedTemporaryFile(delete=False, ...)*، فایل موقت با پسوند مناسب ایجاد می‌شود و تضمین می‌گردد که پس از بسته شدن نیز (*delete=False*) روی دیسک باقی بماند.

۵. **تابع** *process_audio* هدف: تبدیل صوت به متن، شناسایی زبان، ترجمه‌ی آن به انگلیسی (در صورت لزوم)، تولید *embedding* و جمع‌آوری فراداده.

```
def process_audio(temp_path: str) -> Tuple[List[float], dict]:
  """
  Transcribes audio, normalizes text, embeds it,
  and returns (embedding, metadata_dict).
  """
  # 1. Whisper -> Text + Language Detection
  whisper_res = whisper_model.transcribe(temp_path)
  detected_text = whisper_res.get("text", "").strip()
  detected_lang = whisper_res.get("language", "")

  # 2. If Persian -> Translate to English
  if detected_lang.startswith("fa"):
```

```
    processed_text = normalize_text_to_english(detected_text)
  else:
    processed_text = detected_text

  # 3. Embed transcribed text
  q_emb = embed_text(processed_text)

  metadata = {
    "file_name": os.path.basename(temp_path),
    "detected_text": detected_text,
    "processed_text": processed_text,
    "detected_language": detected_lang,
  }

  return q_emb, metadata
```

تعریف تابع و ورودی/خروجی

```
def process_audio(temp_path: str) -> Tuple[List[float], dict]:
```

- **ورودی:** *temp_path: str* – مسیر فایل صوتی (مثل *wav.* یا *mp3.*) که قبلاً روی دیسک ذخیره شده است.
- **خروجی:** یک تاپل (*tuple*) شامل دو مقدار:
  - *q_emb: List[float]* – بردار جاسازی (*embedding*) متن آن فایل صوتی.
  - *metadata: dict* – دیکشنری شامل اطلاعات اضافی درباره‌ی فایل و متن تشخیص‌داده‌شده.

گام‌به‌گام اجرای تابع

۱. **تبدیل صوت به متن با *Whisper* و شناسایی زبان**

```
whisper_res = whisper_model.transcribe(temp_path)
detected_text = whisper_res.get("text", "").strip()
detected_lang = whisper_res.get("language", "")

```

- *whisper_model.transcribe(temp_path)* فایل صوتی را به مدل *Whisper* (*OpenAI Speech Recognition*) ارسال می‌کند تا متن و زبان را استخراج کند.
- *detected_text*: متن استخراج‌شده از صوت.
- *detected_lang*: کد زبان شناسایی‌شده (مانند *"en"* یا *"fa"*).

نتیجه این بخش: صوت تبدیل به متن و شناسایی زبان انجام می‌شود.

۲. **ترجمه به انگلیسی (در صورت فارسی بودن زبان)**

```
if detected_lang.startswith("fa"):
  processed_text = normalize_text_to_english(detected_text)
else:
  processed_text = detected_text

```

- اگر زبان تشخیص‌داده‌شده با *"fa"* شروع شود (فارسی)، تابع *normalize_text_to_english()* فراخوانی می‌شود تا متن به انگلیسی ترجمه گردد.
- این کار متن نهایی را برای تولید *embedding* سازگار با مدل چندمودال (*CLIP*) آماده می‌کند.

نتیجه این بخش: متن نهایی (*processed_text*) به زبان انگلیسی آماده می‌شود.

۳. **گرفتن *embedding* از متن**

```
q_emb = embed_text(processed_text)

```

- تابع *embed_text()* (که قبلاً توضیح داده شد) متن پردازش‌شده را به یک بردار عددی (*embedding*) تبدیل می‌کند که نشان‌دهنده معنای جمله است.

نتیجه این بخش: متن نهایی به فضای برداری (*Vector Space*) نگاشته می‌شود.

۴. **ساخت** *metadata*

```
metadata = {
  "file_name": os.path.basename(temp_path),
  "detected_text": detected_text,
  "processed_text": processed_text,
  "detected_language": detected_lang,
}

```

توضیحات فیلدهای *metadata* دیکشنری متادیتا شامل اطلاعات زیر برای ثبت و استفاده‌ی بعدی است:

- *"file_name"*: نام فایل صوتی.
- *"detected_text"*: متن خام تشخیص داده‌شده توسط مدل *Whisper*.
- *"processed_text"*: متن نهایی بعد از ترجمه به انگلیسی یا نرمال‌سازی (متنی که برای تولید *embedding* استفاده شده است).
- *"detected_language"*: کد زبان شناسایی‌شده.

نتیجه این بخش: همه اطلاعات متنی و زبانی همراه با نام فایل در قالب *metadata* جمع‌آوری می‌شوند.

۵. **برگرداندن خروجی**

```
return q_emb, metadata

```

- خروجی شامل *Embedding* (*q_emb*) برای جستجو در دیتابیس برداری و *Metadata* (*metadata*) برای اطلاعات جانبی است.

۶. **توابع** *get_embedding* **و** *search_by_embedding* هدف: تولید *embedding* (نمایش برداری) برای کوئری ورودی و اجرای جستجوی تشابه برداری (*Vector Similarity Search*) در پایگاه داده *Weaviate*.
این دو تابع در کنار یکدیگر کار می‌کنند تا یک کوئری ورودی (متن یا تصویر) را به بردار تبدیل کرده و نزدیک‌ترین موارد مشابه را از دیتابیس بازیابی کنند.

```
def get_embedding(modality: str, data: str):
  if modality == "text":
    return embed_text(data)
  elif modality == "image":
    return embed_image(data)
  raise ValueError("modality must be 'text' or 'image'")

def search_by_embedding(query_embedding, modality: str, top_k=3):
  try:
    collection = weaviate_client.collections.get(WEAVIATE_COLLECTION_NAME)
    results = collection.query.near_vector(
      near_vector=query_embedding.tolist(),
      limit=top_k,
      filters=weaviate.classes.query.Filter.by_property("modality").equal(
          modality
      ),
    )
    found = []
    if results.objects:
      for obj in results.objects:
          found.append({"properties": obj.properties})
    return found
  except Exception as e:
    print(f"Error searching {modality}: {e}")
    return []
```

تابع *get_embedding(modality: str, data: str)* **وظیفه:** بر اساس نوع داده (*modality*: *"text"* یا *"image"*)، *embedding* مناسب را تولید و برمی‌گرداند.

- *if modality == "text": return embed_text(data)*: اگر نوع داده متنی باشد، متن به یک بردار عددی تبدیل می‌شود.

- *elif modality == "image": return embed_image(data)*: اگر نوع داده تصویری باشد، فایل تصویر به یک *embedding* عددی تبدیل می‌شود.
- *raise ValueError(...) ...*: در صورت نامعتبر بودن *modality*، یک خطا پرتاب می‌شود.

خروجی: یک آرایه‌ی عددی (*numpy array*) شامل *embedding* کوئری ورودی.
تابع *search_by_embedding(query_embedding, modality: str, top_k=3)* **وظیفه:** جستجوی شباهت برداری با استفاده از *embedding* کوئری در پایگاه داده *Weaviate* و بازگرداندن نزدیک‌ترین نتایج.

۱. **اتصال به کالکشن** (*Collection*):

```
collection = weaviate_client.collections.get(WEAVIATE_COLLECTION_NAME)

```

به کالکشن مورد نظر در *Weaviate* که حاوی داده‌های چندمودال است، متصل می‌شود.

۲. **اجرای جستجوی برداری:**

```
results = collection.query.near_vector(
    near_vector=query_embedding.tolist(),
    limit=top_k,
    filters=weaviate.classes.query.Filter.by_property("modality").equal(modality),
)

```

- *collection.query.near_vector*: متد اصلی برای جستجوی تشابه برداری.
- *limit=top_k*: تعداد نتایج مورد نیاز را مشخص می‌کند (به طور پیش‌فرض ۳).
- *filters*: این فیلتر ضروری تضمین می‌کند که جستجو فقط در میان داده‌هایی انجام شود که نوع مدالیته‌ی آن‌ها با کوئری ورودی یکسان است (مثلاً *embedding* یک متن، فقط در میان *embedding* متون جستجو شود).

۳. **استخراج نتایج:**

```
found = []
if results.objects:
    for obj in results.objects:
        found.append({"properties": obj.properties})
return found

```

- داده‌ها و فرادادههای (*metadata*) مرتبط با نزدیک‌ترین بردارها، از فیلد *obj.properties* استخراج شده و در لیست *found* ذخیره می‌شوند.

۴. **مدیریت خطا:**

```
except Exception as e:
  print(f"Error searching {modality}: {e}")
  return []

```

در صورت بروز خطا در حین جستجو در *Weaviate*، پیام خطا چاپ شده و یک لیست خالی برگردانده می‌شود.

۷. **تابع** *normalize_results* هدف: ساده‌سازی خروجی جستجوهای *Weaviate* به یک لیست ساده از ویژگی‌ها (*properties*).

```
def normalize_results(results_list):
  """Converts Weaviate result list to a simple list of properties."""
  out = []
  for r in results_list or []:
    props = r.get("properties") if isinstance(r, dict) else None
    if props:
      out.append(props)
  return out
```

این تابع برای تبدیل نتایج خام دیتابیس به فرمت یکنواخت برای استفاده‌ی آسان‌تر طراحی شده است.

۶.۷ فایل *llm.py*

در نهایت، جهت تولید پاسخ توسط مدل زبان بزرگ (*LLM*)، از تابع *feed_data_into_llm* که در اسکریپت *llm.py* تعریف شده، استفاده می‌گردد.

۱.۶.۷ راه‌اندازی کلاینت *OpenAI*

در ابتدای فایل *llm.py*، کد زیر جهت برقراری ارتباط با مدل‌های زبان بزرگ، تعبیه شده است:

```
# === Initialize OpenAI Client ===
client = OpenAI(
    base_url=LLM_API_BASE,
    api_key=LLM_API_KEY
)
```

این قسمت از کد وظیفه راه‌اندازی (*Initialization*) کلاینت *OpenAI* را بر عهده دارد تا امکان استفاده از مدل‌های زبانی (*LLM*ها) مانند *GPT* فراهم گردد. متغیر *client* به عنوان رابط اصلی برای تعامل با *LLM* عمل می‌کند.

۱. *client = OpenAI(...)*: در این خط، یک شیء (*object*) از کلاس *OpenAI* ایجاد شده و در متغیر *client* ذخیره می‌شود. این شیء در واقع کلاینتی است که از طریق آن می‌توان درخواست‌ها را به مدل‌های *OpenAI* یا سرورهای سازگار با *API* آن (مانند سرورهای محلی یا *third-party LLM servers*) ارسال نمود.

۲. پارامترهای کلیدی:

- *base_url=LLM_API_BASE*: این پارامتر آدرس *URL* سروری که درخواست‌ها به آن ارسال می‌شود را مشخص می‌کند.
  - در استفاده از *API* رسمی *OpenAI*: *LLM_API_BASE = "https://api.openai.com/v1"*.
  - در صورت استفاده از مدل‌های محلی (*local*) یا میزبانی‌شده توسط کاربر (*self-hosted* مانند *Ollama*، *LM Studio*، یا *vLLM*): *LLM_API_BASE* می‌تواند آدرس *API* محلی باشد، مثلاً: *"http://localhost:8000/v1"*.
- *api_key=LLM_API_KEY*: این کلید امنیتی جهت احراز هویت (*authentication*) کاربر در سرور الزامی است. مقدار آن معمولاً یک رشته محرمانه مانند *"sk-xxxx..."* است.

۳. نتیجه: پس از اجرای این خطوط، متغیر *client* به یک شیء فعال از نوع *OpenAI* تبدیل می‌شود.

لازم به ذکر است که مقادیر *LLM_API_BASE* و *LLM_API_KEY* از فایل تنظیمات *config.py* بارگذاری می‌شوند.

۲.۶.۷ تشریح تابع *feed_data_into_llm*

```
def feed_data_into_llm(llm_data: dict) -> str:
    """Send multimodal data (text + image + audio) to OpenRouter model."""

    def section(title: str, content: str) -> str:
        return f"\n\n### {title}\n{content.strip()}"

    sections = []

    # --- User Text ---
    if llm_data.get("user_text_query"):
        sections.append(section("User Text Query", llm_data["user_text_query"]))

    # --- User Images ---
    image_base64s = []
    if llm_data.get("user_image_queries"):
        for img in llm_data["user_image_queries"]:
            pil_img = to_pil_image(img)
            if pil_img:
                pil_img = pil_img.resize((128, 128), Image.LANCZOS)
                image_base64s.append(image_to_base64_data_url(pil_img))
        sections.append(section("User Image Queries",
        f"{len(image_base64s)} image(s) attached." if image_base64s else
        "<no usable images>"))

    # --- User Audio ---
    if llm_data.get("user_audio_queries"):
        aud_list = "\n\n".join(
            [f"Audio {i+1} (Transcription):\n{t}" for i, t in
            enumerate(llm_data["user_audio_queries"])]
```

```
        )
        sections.append(section("User Audio Queries", aud_list))

    # --- Retrieved Texts ---
    if llm_data.get("retrieved_texts"):
        txt_list = "\n\n".join(
            [f"Retrieved Text {i+1}:\n{t}" for i, t in
            enumerate(llm_data["retrieved_texts"])]
        )
        sections.append(section("Retrieved Texts", txt_list))

    # --- Retrieved Images ---
    retrieved_image_base64s = []
    print("Retrieved images count:", len(llm_data.get("retrieved_images", [])))
    if llm_data.get("retrieved_images"):
        for img in llm_data["retrieved_images"]:
            pil_img = to_pil_image(img)
            if pil_img:
                pil_img = pil_img.resize((128, 128), Image.LANCZOS)
                retrieved_image_base64s.append(image_to_base64_data_url(pil_img))
                print("Retrieved image converted successfully")

        sections.append(section("Retrieved Images",
        f"{len(retrieved_image_base64s)} image(s) attached." if
        retrieved_image_base64s else "<no usable retrieved images>"))

    # --- Retrieved Audios ---
    if llm_data.get("retrieved_audios"):
        raud_list = "\n\n".join(
            [f"Retrieved Audio {i+1} (Transcription):\n{t}" for i, t in
            enumerate(llm_data["retrieved_audios"])]
        )
        sections.append(section("Retrieved Audio Transcriptions", raud_list))

    # --- Merge all context ---
    full_context = "\n\n".join(sections).strip() or "No multimodal content provided."

    # === Construct message content ===
    content_list = [{"type": "text", "text": full_context}]

    # Add images as base64 URLs
    for img_url in image_base64s + retrieved_image_base64s:
        content_list.append({
            "type": "image_url",
            "image_url": {"url": img_url}
        })

    # === Call the LLM ---
    try:
        completion = client.chat.completions.create(
            model=LLM_MODEL_NAME,   # e.g. "openai/gpt-5-image-mini"
            extra_headers={
                "HTTP-Referer": "http://localhost",
                "X-Title": "Multimodal RAG Chatbot"
            },
            messages=[
                {
                    "role": "system",
                    "content": (
                        "You are a multimodal reasoning assistant.\n"
                        "You receive user inputs and retrieved multimodal data (text,
    image, audio).\n"
                        "If the user asked a question, answer it using relevant data.\n"
                        "If not, describe the multimodal inputs clearly in Markdown."
                    )
                },
                {
                    "role": "user",
                    "content": content_list
                }
            ]
        )
```

```
        response_text = completion.choices[0].message.content
    except Exception as e:
        response_text = f"LLM call failed: {e}"

    # === Logging ===
    log_dir = os.path.join(PROJECT_ROOT_PATH, "log")
    os.makedirs(log_dir, exist_ok=True)
    log_path = os.path.join(log_dir, "llm_full_context.txt")
    try:
        with open(log_path, "w", encoding="utf-8") as f:
            f.write("=== FULL CONTEXT SENT TO LLM ===\n\n")
            f.write(full_context + "\n\n")
            f.write("=" * 50 + "\n\n")
            f.write("=== LLM RESPONSE ===\n\n")
            f.write(response_text + "\n")
        print(f"Full context written to {log_path}")
    except Exception as e:
        print(f"Failed to write LLM log: {e}")

    return response_text
```

این تابع *feed_data_into_llm()* یکی از مؤلفه‌های اصلی در معماری یک سیستم *Multimodal RAG Chatbot* محسوب می‌شود. وظیفه آن تجمیع، آماده‌سازی و ارسال تمامی داده‌های چندمودالی (شامل متن، تصویر، و صوت) به مدل زبان بزرگ (*LLM*) و بازگرداندن پاسخ نهایی مدل است.

تعریف تابع *def feed_data_into_llm(llm_data: dict) -> str:*

- ورودی: یک دیکشنری (*dictionary*) به نام *llm_data* که حاوی داده‌های چندمودالی پروژه است (این دیکشنری در تابع *search_multimodal* تولید می‌شود).
- خروجی: یک رشته (*string*) که شامل پاسخ نهایی تولید شده توسط *LLM* است.

تابع کمکی داخلی: *section()* *def section(title: str, content: str) -> str:*

- این تابع به منظور قالب‌بندی محتوا در فرمت *Markdown* طراحی شده است.
- از آن برای تفکیک و ارائه منظم ورودی‌ها به *LLM* با استفاده از عنوان و محتوا استفاده می‌شود.

تشریح مراحل آماده‌سازی داده

۱. **ورودی متنی کاربر** (*user_text_query*): در صورتی که ورودی متنی از سوی کاربر دریافت شده باشد، این متن به عنوان یک مؤلفه مجزا در لیست *sections* ثبت می‌گردد.

```
if llm_data.get("user_text_query"):
    sections.append(section("User Text Query", llm_data["user_text_query"]))

```

۲. **تصاویر کاربر** (*user_image_queries*):

- تصاویر ورودی به تابع *to_pil_image()* ارسال و به فرمت *PIL* تبدیل می‌گردند. تصاویر ورودی کاربر از طریق تابع *to_pil_image()* به فرمت استاندارد *PIL* (*Python Imaging Library*) تبدیل می‌شوند.
- جهت کاهش حجم ورودی، تصاویر به ابعاد 128×128 تغییر اندازه داده می‌شوند. برای بهینه‌سازی بار پردازشی و کاهش تأخیر، ابعاد تصاویر ورودی به صورت یکنواخت به 128×128 پیکسل (با استفاده از متد *Image.LANCZOS*) تغییر اندازه می‌یابد.
- تصاویر با استفاده از *image_to_base64_data_url()* به رشته‌ی *Base64* تبدیل می‌شوند تا قابلیت ارسال در بدنه *JSON* به مدل را داشته باشند. سپس، تصاویر به کد *Base64* تبدیل می‌شوند تا بتوان آن‌ها را در قالب *JSON* و مطابق با الزامات *API* مدل به *LLM* ارسال نمود.

```
image_base64s = []
if llm_data.get("user_image_queries"):
    for img in llm_data["user_image_queries"]:
        pil_img = to_pil_image(img)
        if pil_img:
            pil_img = pil_img.resize((128, 128), Image.LANCZOS)
            image_base64s.append(image_to_base64_data_url(pil_img))
```

```
    sections.append(section("User Image Queries", f"{len(image_base64s)} image(s)
    attached." if image_base64s else "<no usable images>"))

```

۳. **صوت کاربر** (*user_audio_queries*): در صورت وجود ورودی صوتی، فرض بر این است که داده‌های صوتی قبلاً به متن (*Transcription*) تبدیل شده‌اند و ترنسکریپت‌های حاصله در قالب *Markdown* به منظور ورود به *LLM* آماده‌سازی می‌گردند.

```
if llm_data.get("user_audio_queries"):
    aud_list = "\n\n".join(
        [f"Audio {i+1} (Transcription):\n{t}" for i, t in
    enumerate(llm_data["user_audio_queries"])]
    )
    sections.append(section("User Audio Queries", aud_list))

```

۴. **متون بازیابی‌شده** (*RAG*) (*retrieved_texts*): متون مرتبطی که از طریق سیستم بازیابی اطلاعات (*Retrieval*) (مانند *Weaviate*) از پایگاه داده به دست آمده‌اند، به عنوان زمینه (*context*) جهت افزایش دقت و مرتبط بودن پاسخ *LLM*، به لیست بخش‌ها افزوده می‌گردند. این متون در واقع *context retrieval* محسوب می‌شوند.

```
if llm_data.get("retrieved_texts"):
    txt_list = "\n\n".join(
        [f"Retrieved Text {i+1}:\n{t}" for i, t in
    enumerate(llm_data["retrieved_texts"])]
    )
    sections.append(section("Retrieved Texts", txt_list))

```

۵. **تصاویر بازیابی‌شده** (*retrieved_images*): تصاویر مرتبط بازیابی‌شده از پایگاه داده، مشابه تصاویر کاربر، پس از تغییر اندازه و تبدیل به *Base64*، به ورودی نهایی مدل اضافه می‌شوند.

```
retrieved_image_base64s = []
for img in llm_data.get("retrieved_images", []):
    pil_img = to_pil_image(img)
    if pil_img:
        pil_img = pil_img.resize((128, 128), Image.LANCZOS)
        retrieved_image_base64s.append(image_to_base64_data_url(pil_img))
sections.append(section("Retrieved Images", f"{len(retrieved_image_base64s)}
    image(s) attached." if retrieved_image_base64s else "<no usable retrieved
    images>"))

```

۶. **صوت‌های بازیابی‌شده** (*retrieved_audios*): ترنسکریپت‌های صوتی بازیابی‌شده به متن نهایی افزوده می‌شوند تا *LLM* بتواند از اطلاعات شنیداری مرتبط با جستجو نیز استفاده نماید.

```
if llm_data.get("retrieved_audios"):
    raud_list = "\n\n".join(
        [f"Retrieved Audio {i+1} (Transcription):\n{t}" for i, t in
    enumerate(llm_data["retrieved_audios"])]
    )
    sections.append(section("Retrieved Audio Transcriptions", raud_list))

```

ترکیب داده‌ها و فراخوانی *LLM*

- **ترکیب زمینه**: تمامی بخش‌های آماده‌سازی شده از ورودی‌های کاربر و داده‌های بازیابی‌شده به یک رشته واحد (*full_context*) تلفیق می‌گردند تا به عنوان ورودی متنی جامع به مدل ارسال شوند.

```
full_context = "\n\n".join(sections).strip() or "No multimodal content provided."

```

- **ساخت پیام**: لیست پیام‌های ارسالی به *LLM* شامل *full_context* به عنوان متن و رشته‌های *Base64* تصاویر (شامل تصاویر کاربر و تصاویر بازیابی‌شده) است.

```
content_list = [{"type": "text", "text": full_context}]
for img_url in image_base64s + retrieved_image_base64s:
    content_list.append({"type": "image_url", "image_url": {"url": img_url}})

```

- **فراخوانی مدل**: درخواست به مدل زبانی از طریق تابع *client.chat.completions.create* ارسال می‌شود و شامل دو پیام اصلی است: پیام *system* (جهت تعریف نقش دستیار) و پیام *user* (شامل *content_list* که حاوی داده‌های چندمودالی است).

```
completion = client.chat.completions.create(
    model=LLM_MODEL_NAME,
    extra_headers={
        "HTTP-Referer": "http://localhost",
        "X-Title": "Multimodal RAG Chatbot"
    },
    messages=[
        {
            "role": "system",
            "content": (
                "You are a multimodal reasoning assistant.\n"
                "You receive user inputs and retrieved multimodal data (text, image, audio).\n"
                "If the user asked a question, answer it using relevant data.\n"
                "If not, describe the multimodal inputs clearly in Markdown."
            )
        },
        {
            "role": "user",
            "content": content_list
        }
    ]
)

```

- **پاسخ**: مدل بر اساس داده‌های ورودی، پاسخ نهایی را تولید می‌کند: *response_text = completion.choices[0].message.content*.

مکانیسم ثبت لاگ (*Logging*) کل ورودی (*full_context*) و خروجی (*response_text*)، به منظور اشکال‌زدایی و مستندسازی جریان داده، در فایل *llm_full_context.txt* واقع در پوشه *log* ذخیره می‌گردند.

```
log_path = os.path.join(PROJECT_ROOT_PATH, "log", "llm_full_context.txt")
```

نتیجه نهایی تابع، پاسخ نهایی تولیدشده توسط *LLM* (*response_text*) را بازمی‌گرداند تا جهت نمایش در واسط کاربری (*UI*) یا محیط چت‌بات (*Chatbot*) مورد استفاده قرار گیرد.

```
return response_text
```

۳.۶.۷ توابع کمکی

در تابع *feed_data_into_llm* از توابع زیر استفاده شده است:

تابع *to_pil_image*

```
def to_pil_image(img_candidate):
    """Convert input to a PIL.Image (RGB)."""
    if img_candidate is None:
        return None
    if isinstance(img_candidate, Image.Image):
        return img_candidate.convert("RGB")
    if isinstance(img_candidate, (bytes, bytearray)):
        try:
            return Image.open(io.BytesIO(img_candidate)).convert("RGB")
        except Exception:
            return None
    if isinstance(img_candidate, str):
        if os.path.exists(img_candidate):
            try:
```

```
                return Image.open(img_candidate).convert("RGB")
            except Exception:
                return None
        if img_candidate.startswith("http://") or img_candidate.startswith("https://"):
            try:
                resp = requests.get(img_candidate, timeout=5)
                if resp.status_code == 200:
                    return Image.open(io.BytesIO(resp.content)).convert("RGB")
            except Exception:
                return None
        return None
    if isinstance(img_candidate, dict):
        b = img_candidate.get("bytes") or img_candidate.get("content") or
    img_candidate.get("data")
        if isinstance(b, (bytes, bytearray)):
            try:
                return Image.open(io.BytesIO(b)).convert("RGB")
            except Exception:
                return None
        path = img_candidate.get("path") or img_candidate.get("file")
        if isinstance(path, str) and os.path.exists(path):
            try:
                return Image.open(path).convert("RGB")
            except Exception:
                return None
        return None
    return None
```

تابع *to_pil_image(img_candidate)* یک **تبدیل‌کننده‌ی همه‌منظوره** است که تلاش می‌کند هر ورودی تصویری ممکن (شامل فایل‌ها، بایت‌ها، *URL*ها، و داده‌های ساختاریافته) را به یک شیء تصویر از نوع *PIL.Image* (در حالت رنگی *RGB*) تبدیل نماید.

خلاصه عملکرد *to_pil_image* مرحله‌به‌مرحله

حالت ورودی	عملکرد تابع
None	مقدار *None* بازگردانده می‌شود.
نوع *PIL.Image*	تصویر ورودی به حالت رنگی *"RGB"* تبدیل شده و بازگردانده می‌شود.
نوع *bytes* یا *bytearray*	تلاش می‌شود تصویر از داده‌ی باینری بارگذاری شود (*Image.open(io.BytesIO(...))*).
نوع *str* (رشته)	• اگر مسیر فایل (*path*) باشد : تصویر از فایل محلی بارگذاری می‌شود. • اگر *URL* باشد : تصویر از طریق شبکه دانلود و باز می‌شود.
نوع *dict*	داده‌های تصویر از کلیدهای *"bytes"*، *"data"*، *"content"*، یا *"path"* استخراج و شیء تصویر ساخته می‌شود.
سایر انواع	مقدار *None* بازگردانده می‌شود.

نتیجه: این تابع به عنوان یک رابط استانداردساز عمل می‌کند تا تمامی فرمت‌های ورودی تصویر برای پردازش‌های بعدی در سیستم، یکپارچه‌سازی شوند.

تابع *image_to_base64_data_url*

```
def image_to_base64_data_url(pil_img):
    """Convert PIL.Image to base64 data URL (JPEG)."""
    buffer = io.BytesIO()
    pil_img.save(buffer, format="JPEG")
    img_str = base64.b64encode(buffer.getvalue()).decode("utf-8")
    return f"{img_str}"
```

این تابع تصویر ورودی (*PIL.Image*) را به یک رشته‌ی *Base64 Data URL* (*...*) تبدیل می‌کند که استاندارد الزامی برای ارسال داده‌های تصویری به مدل‌های چندمودالی از طریق *API* است.

تشریح مفهوم *Base64* *Base64* یک روش کدگذاری است که داده‌های باینری (نظیر تصاویر) را به رشته‌های متنی *ASCII* تبدیل می‌کند. این فرآیند حیاتی است زیرا مدل‌های زبان بزرگ (*LLM*ها) و اکثر *API*ها، داده‌ها را به طور عمده با فرمت متنی (*string*) پردازش و منتقل می‌نمایند.

لزوم تبدیل به *Base64*

- **انتقال امن**: *LLM* یا *API* توانایی پردازش مستقیم فایل‌های باینری (مانند *jpg.* یا *png.*) را در ساختار پیام‌های *JSON* ندارند.
- **استانداردسازی**: با تبدیل به *Base64*، تصویر به صورت یک رشته‌ی متنی ایمن و قابل انتقال در پروتکل‌های تحت وب در می‌آید.
- **قابلیت بازیابی**: در سمت مقصد (مدل یا سرور)، داده‌های *Base64* دوباره به محتوای باینری تصویر تبدیل و جهت پردازش استفاده می‌شوند.