Special Topics in Artificial Intelligence

Midterm Project

HistoryLens: Multimodal Historical Landmark Guide

Kaan Yalman

Student ID: 2104371

Contents

1 Explanation of Project

1.1 Project Overview

The **HistoryLens** project aims to develop a multimodal historical landmark guide that allows users to identify and learn about historical buildings through computer vision and generative AI. The core functionality enables a user to take a photograph of a historical structure, specifically focusing on landmarks in Istanbul and Turkey (e.g., Kız Kulesi), and receive a detailed audio-visual summary of the building's history.

1.2 Visual Recognition Model

To identify the buildings, the system utilizes a **Vision Transformer (ViT)** model. Due to the scarcity of large, labeled datasets for specific historical landmarks, a **one-shot learning** approach was adopted. This method allows the model to classify images correctly with only a single reference example per class.

The dataset consists of 11 distinct classes representing iconic historical sites. One reference image was collected for each class using a custom Python script to scrape Google Images. The specific classes included in the model are:

- Rumeli Hisarı
- Topkapı Sarayı
- Kız Kulesi
- Sultan Ahmet Camii
- Galata Kulesi
- Mevlana Müzesi
- Anıtkabir
- Sümela Manastırı
- İzmir Saat Kulesi
- Ayasofya Camii
- Yerebatan Sarnıcı

1.3 Methodology

The recognition process is divided into two phases:

1. **Reference Phase (Training):** The model captures and saves the vector embeddings of the single reference image (one-shot) for each of the 11 classes.
2. **Inference Phase (Testing):** To validate the system, a test set containing 3 samples per class was collected. During inference, the model generates embeddings for the input image and compares them against the stored reference embeddings to predict the correct building class.

1.4 RAG and Multimodal Interaction

Once the building is identified, the system employs a **Retrieval-Augmented Generation (RAG)** pipeline to provide context. Historical explanations were scraped from Wikipedia and stored in a structured JSON format.

When a building is detected, the RAG system retrieves the relevant historical text. A Large Language Model (LLM) then processes this text to generate a concise summary, covering key details such as the construction date, the builder, and historical significance.

Finally, the summarized text is processed by a Text-to-Speech (TTS) audio model. The system includes a user interface that allows the user to select their preferred language (Turkish or English), and the audio model vocalizes the summary accordingly.

2 Motivation

The primary motivation behind the development of **HistoryLens** is to revolutionize the way tourists interact with cultural heritage. Traditionally, visitors to historical sites often rely heavily on human tour guides or physical guidebooks. While valuable, these resources can be restrictive regarding scheduling, language availability, and the specific depth of information provided.

This project aims to bridge that gap by offering an independent, accessible, and AI-powered alternative. By leveraging advanced computer vision and multimodal technologies, the goal is to empower tourists to explore historical landmarks at their own pace. Instead of being strictly bound to a tour group, users can utilize this tool to instantly identify buildings and receive comprehensive, vocalized historical narratives on demand. Ultimately, this project seeks to make history more accessible, engaging, and personalized for travelers exploring Turkey's rich cultural landscape.

3 Code Part

3.1 Environment Setup and Initialization

The project was implemented using a **Google Colab Pro** environment. To ensure efficient inference times, particularly for the Vision Transformer (ViT) and LLM components, the runtime was configured with **High RAM** and an **NVIDIA A100 GPU**.

Figure 1 demonstrates the initial setup of the environment across the first two code cells.

1. Library Installation (Cell 1): The process begins by installing the necessary dependencies:

- **Transformers (Hugging Face):** Essential for loading pre-trained models such as ViT and the summarization LLM.
- **PyTorch (torch):** The core framework used for GPU acceleration and vector operations (specifically cosine similarity).
- **Pillow (PIL) & OpenCV:** Used for opening and preprocessing image files.
- **gTTS:** Google Text-to-Speech library for vocalizing the summaries.
- **Accelerate:** Optimized library to speed up model inference.

2. Imports and Data Loading (Cell 2): Following the installation, the necessary libraries are imported. Crucially, this cell handles the data pipeline setup. Google Drive is mounted to access the raw dataset. To optimize training and inference speed, the script checks for the dataset directory and explicitly copies it from Drive to the local Colab runtime environment (‘/content/dataset‘). This step significantly reduces file I/O latency compared to reading directly from Drive.

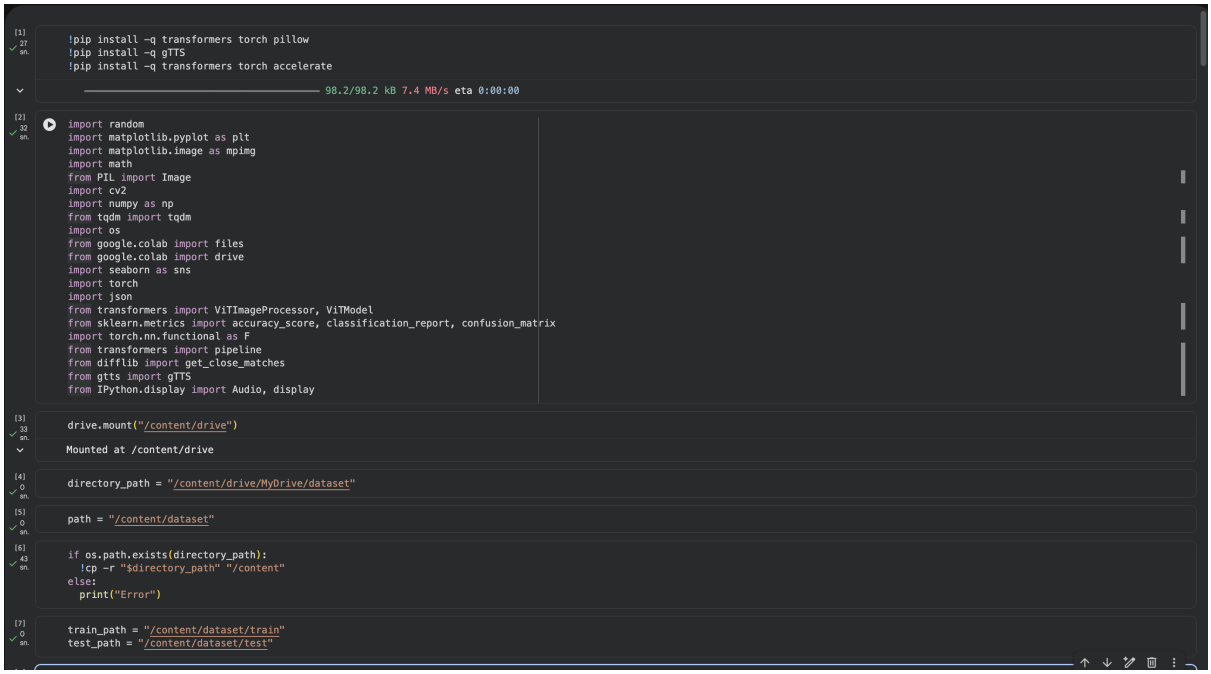


Figure 1: Initialization of the Colab environment: installing dependencies, importing libraries, and moving the dataset to the local runtime.

3.2 Data Preparation and Cleaning

After extracting the dataset, the code defines the specific file paths for the training and testing sets. As seen in Figure 2, the initial traversal of the directories using ‘os.listdir‘ revealed the presence of system artifacts, specifically `.DS_Store` files (common in macOS environments), which are irrelevant to the training process and can cause errors during iteration.

To ensure a clean list of class labels, a filtering step was implemented. As shown in Figure 3, a list comprehension is used to iterate through the directory contents and exclude any file starting with a dot (‘.‘). This results in a clean list of the 11 target classes (e.g., ’yerebatan_sarnici’, ’izmir_saat_kulesi’, etc.).

```
train_path = "/content/dataset/train"
test_path = "/content/dataset/test"
```

```
os.listdir(train_path)
```

```
['yerebatan_sarnici',
 'izmir_saat_kulesi',
 'rumeli_hisari',
 'galata_kulesi',
 'mevlana_muzesi',
 'anitkabir',
 '.DS_Store',
 'sultan_ahmet_camii',
 'kiz_kulesi',
 'sumela_manastiri',
 'ayasofya_camii',
 'topkapi_sarayi']
```

```
os.listdir(test_path)
```

```
['yerebatan_sarnici',
 'izmir_saat_kulesi',
 'rumeli_hisari',
 'galata_kulesi',
 'mevlana_muzesi',
 'anitkabir',
 '.DS_Store',
 'sultan_ahmet_camii',
 'kiz_kulesi',
 'sumela_manastiri',
 'ayasofya_camii',
 'topkapi_sarayi']
```

```
train_list = os.listdir(train_path)
```

```
train_list
```

```
['yerebatan_sarnici',
 'izmir_saat_kulesi',
 'rumeli_hisari',
 'galata_kulesi',
 'mevlana_muzesi',
 'anitkabir',
 '.DS_Store',
 'sultan_ahmet_camii',
 'kiz_kulesi',
 'sumela_manastiri',
 'ayasofya_camii',
 'topkapi_sarayi']
```

Figure 2: Defining training and testing paths and identifying system artifacts (.DS_Store) in the directory list.

```
train_list = [d for d in os.listdir(train_path) if not d.startswith(".")]
train_list
```

```
['yerebatan_sarnici',
 'izmir_saat_kulesi',
 'rumeli_hisari',
 'galata_kulesi',
 'mevlana_muzesi',
 'anitkabir',
 'sultan_ahmet_camii',
 'kiz_kulesi',
 'sumela_manastiri',
 'ayasofya_camii',
 'topkapi_sarayi']
```

```
test_list = [d for d in os.listdir(test_path) if not d.startswith(".")]
test_list
```

```
['yerebatan_sarnici',
 'izmir_saat_kulesi',
 'rumeli_hisari',
 'galata_kulesi',
 'mevlana_muzesi',
 'anitkabir',
 'sultan_ahmet_camii',
 'kiz_kulesi',
 'sumela_manastiri',
 'ayasofya_camii',
 'topkapi_sarayi']
```

```
len(train_list)
```

```
11
```

Figure 3: Filtering the class lists to remove hidden system files and verifying the final class count (11 classes).

3.3 Dataset Visualization

Before proceeding to model training, it is crucial to verify the integrity of the image data.

First, a single sample image (Yerebatan Sarnıcı) was loaded and displayed to confirm that the file paths were constructed correctly (Figure 4).

Subsequently, to validate the entire dataset, a visualization grid was generated. The code iterates through the cleaned 'train_list', loads the first image from each class directory, and displays them in a subplot grid. This provides a visual confirmation that all 11 classes are correctly accessible and labeled (Figure 5).

Figure 4: Visualizing a single sample (Yerebatan Sarnıcı) to verify path correctness.

```python
plt.figure(figsize=(15, 10))

for i in range(11):
    if i < len(train_list):
        image_name = train_list[i]
        image_path = os.path.join(train_path, image_name, "000001.jpg")
        img_data = mpimg.imread(image_path)

        image_name = image_name.replace("_", " ").title()
        plt.subplot(3, 4, i + 1)
        plt.imshow(img_data)
        plt.title(f"Label: {image_name}")
        plt.axis("off")

plt.tight_layout()
plt.show()
```

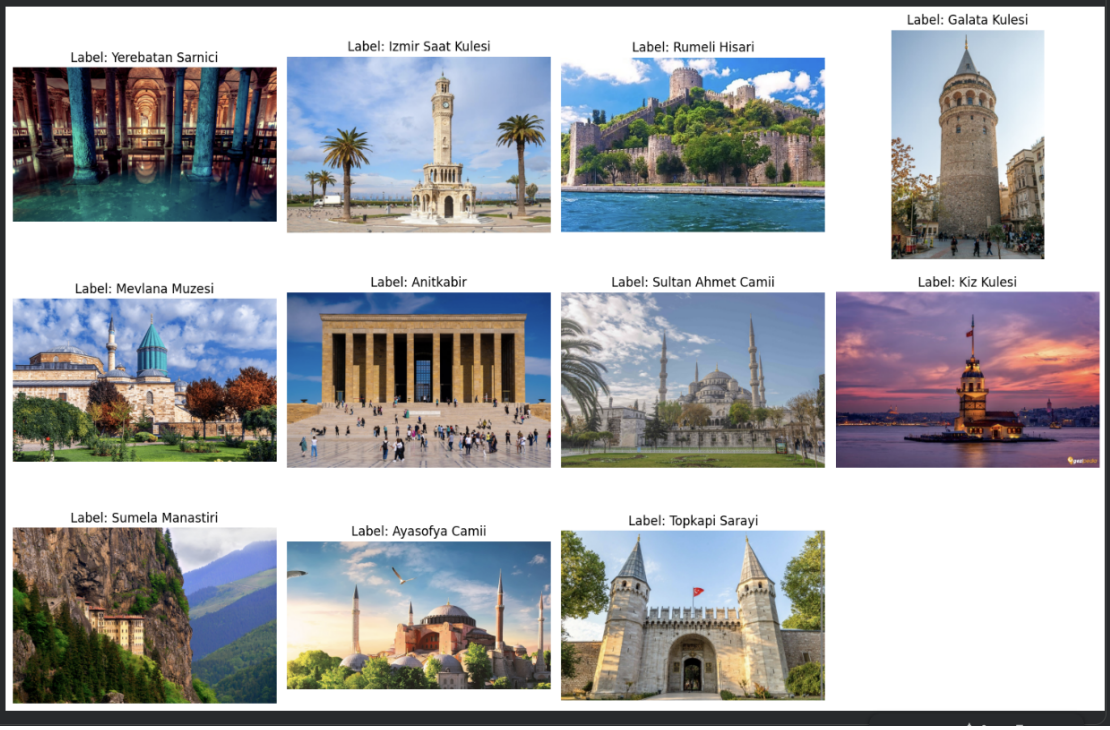


Figure 5: Grid visualization displaying one representative sample from each of the 11 historical landmark classes.

3.4 Image Preprocessing

To prepare the dataset for the Vision Transformer (ViT) model, the images must be standardized to a specific resolution. As shown in Figure 6, a preprocessing pipeline was implemented to resize all images to 224×224 pixels, which is the expected input dimension for standard pre-trained ViT architectures.

The ‘preprocess_image‘ function utilizes the **OpenCV** library (‘cv2‘). Specifically, the `cv2.INTER_LANCZOS4` interpolation method was selected for resizing. Lanczos interpolation is generally preferred over standard bilinear or nearest-neighbor methods as it preserves high-frequency structural details, which are critical for distinguishing between architectural landmarks.

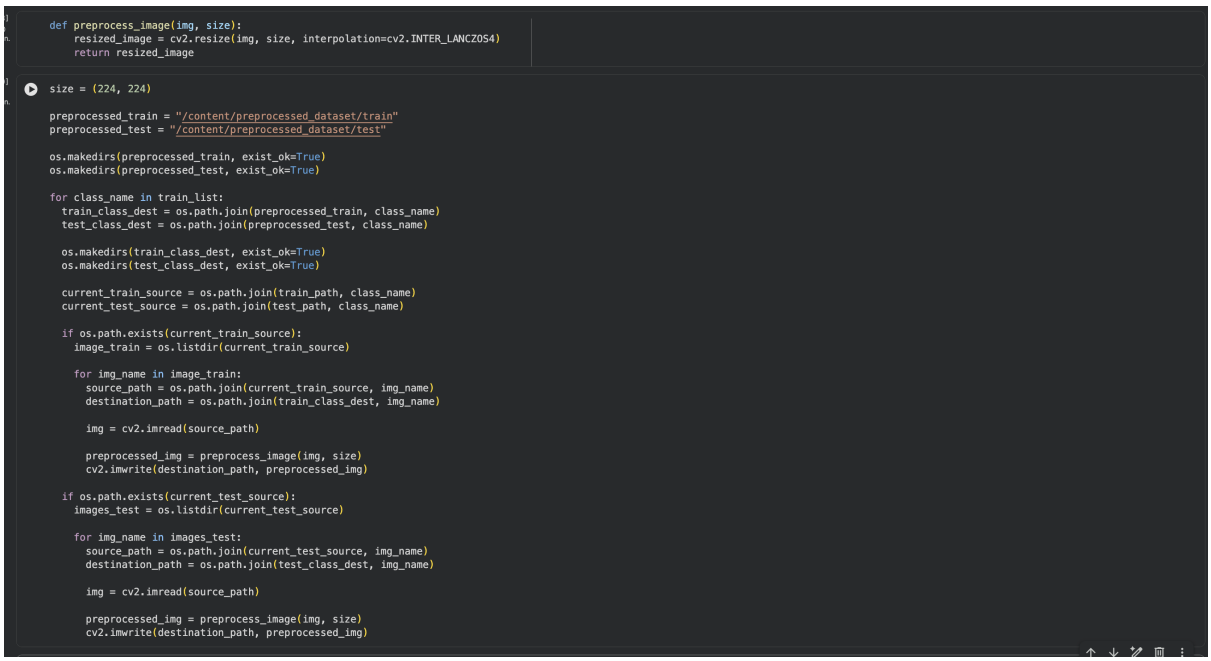
The script automates this process by:

1. Creating a parallel directory structure (‘/content/preprocessed_dataset‘) for both

training and testing sets using the ‘os‘ library.

2. Iterating through every class folder and image file.
3. Applying the resizing function to each image.
4. Saving the processed images to the new destination paths using ‘cv2.imwrite‘.

While more advanced data augmentation techniques (such as rotation or color jittering) could be applied to further improve robustness, this project focuses on a streamlined pipeline where high-quality resizing is the primary preprocessing step.



```python
def preprocess_image(img, size):
    resized_image = cv2.resize(img, size, interpolation=cv2.INTER_LANCZOS4)
    return resized_image

size = (224, 224)

preprocessed_train = "/content/preprocessed_dataset/train"
preprocessed_test = "/content/preprocessed_dataset/test"

os.makedirs(preprocessed_train, exist_ok=True)
os.makedirs(preprocessed_test, exist_ok=True)

for class_name in train_list:
  train_class_dest = os.path.join(preprocessed_train, class_name)
  test_class_dest = os.path.join(preprocessed_test, class_name)

  os.makedirs(train_class_dest, exist_ok=True)
  os.makedirs(test_class_dest, exist_ok=True)

  current_train_source = os.path.join(train_path, class_name)
  current_test_source = os.path.join(test_path, class_name)

  if os.path.exists(current_train_source):
    image_train = os.listdir(current_train_source)

    for img_name in image_train:
      source_path = os.path.join(current_train_source, img_name)
      destination_path = os.path.join(train_class_dest, img_name)

      img = cv2.imread(source_path)

      preprocessed_img = preprocess_image(img, size)
      cv2.imwrite(destination_path, preprocessed_img)

  if os.path.exists(current_test_source):
    images_test = os.listdir(current_test_source)

    for img_name in images_test:
      source_path = os.path.join(current_test_source, img_name)
      destination_path = os.path.join(test_class_dest, img_name)

      img = cv2.imread(source_path)

      preprocessed_img = preprocess_image(img, size)
      cv2.imwrite(destination_path, preprocessed_img)
```

Figure 6: Implementation of the preprocessing pipeline: resizing images to 224×224 using Lanczos interpolation and saving them to a new directory structure.

3.5 Model Initialization and One-Shot Strategy

With the preprocessed dataset ready, the system initializes the Vision Transformer (ViT) model. As shown in Figure 7, the directory paths are updated to point to the new `preprocessed_dataset`.

The model implementation relies on the **Hugging Face Transformers** library. The specific architecture selected is `google/vit-base-patch16-224`, a standard ViT model pre-trained on ImageNet.

3.5.1 Device Configuration

To leverage the hardware acceleration provided by Colab Pro, the code checks for CUDA availability. The model is explicitly moved to the GPU (‘device = ’cuda’‘) to ensure rapid inference.

3.5.2 Feature Extraction Mode

A critical step in this implementation is setting the model to evaluation mode using ‘model.eval()‘.

- **No Training:** Unlike traditional supervised learning, we are *not* updating the model’s weights (backpropagation).
- **One-Shot Mechanism:** Instead, the model acts as a fixed feature extractor. It calculates the vector embedding for the single reference image of each class.
- **Similarity Matching:** This approach is analogous to modern FaceID systems. Just as a phone stores a mathematical representation of your face to match against future scans, this system stores the ”embedding” of a historical building. During the test phase, the system will calculate the Cosine Similarity between the input image and these stored embeddings to find the closest match.

Figure 7: Loading the pre-trained ViT model, moving it to the GPU, and setting it to evaluation mode for feature extraction.

3.6 Reference Embedding Extraction

The core of the one-shot learning mechanism involves creating a database of reference vectors. As shown in Figure 8, the code iterates through each class in the training directory to process the single reference image available for that landmark.

The feature extraction pipeline consists of the following steps:

1. **Image Loading and Conversion:** Each image is loaded using PIL and explicitly converted to RGB format (‘.convert(”RGB”)‘). This step ensures consistency, preventing errors that might arise from grayscale images or PNG files containing a fourth alpha (transparency) channel, which the model cannot process.

2. **Preprocessing:** The images are passed through the ViT processor to be normalized and converted into PyTorch tensors compatible with the GPU.

3. **Inference (No Gradient):** The forward pass is executed within a ‘torch.no$_grad()‘context block.S$

3. **Vector Extraction:** The model outputs the hidden states for the image patches. The code extracts the embedding corresponding to the first token (index 0), accessed via ‘outputs.last$_hidden_state[:, 0, :]‘.InVisionTransformers, thisspecificvectorservesasthedense$

3. **Storage:** The resulting embedding vector is moved to the CPU and stored in the ‘reference$_database‘dictionary, keyedbytheclassname.Finally, thisdictionaryisserializedandsave$

```
reference_database = {}

for class_name in class_names:
  class_path = os.path.join(train_dir, class_name)

  images = os.listdir(class_path)

  for img in images:
    img_path = os.path.join(class_path, img)

    image = Image.open(img_path).convert("RGB")
    inputs = processor(images=image, return_tensors="pt").to(device)

    with torch.no_grad():
      outputs = model(**inputs)

    embedding = outputs.last_hidden_state[:, 0, :].cpu()
    reference_database[class_name] = embedding

torch.save(reference_database, output_file)
```

Figure 8: Iterating through training classes, extracting feature embeddings from the ViT model, and saving them to a reference database.

3.7 Inference Phase and Similarity Matching

Once the reference database is established, the system proceeds to the inference phase to evaluate its performance on the unseen test dataset. As demonstrated in the code snippet in Figure 9, the process iterates through each image in the test directory.

For every test image, the exact same preprocessing pipeline used in the reference phase is applied: loading the image, converting it to RGB, and extracting its feature embedding vector using the frozen ViT model. This resulting vector is termed the "query vector."

The core classification logic relies on **Cosine Similarity**. The system performs an exhaustive comparison, calculating the cosine similarity score between the query vector and every stored reference vector in the database. The reference class that yields the highest similarity score is assigned as the predicted label for the test image.

$$similarity(A, B) = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} \tag{1}$$

Where A is the query vector and B is a reference vector. This process is computationally intensive, as it requires iterating through the entire database for every single test image, leading to noticeable inference times.

```python
y_true = []
y_pred = []
classes = sorted(list(reference_database.keys()))

for class_name in classes:
  class_path = os.path.join(test_dir, class_name)

  images = os.listdir(class_path)
  for image in images:

    img_path = os.path.join(class_path, image)


    image = Image.open(img_path).convert("RGB")
    inputs = processor(images=image, return_tensors="pt").to(device)

    with torch.no_grad():
      outputs = model(**inputs)

      query_vec = outputs.last_hidden_state[:, 0, :].view(-1)

      best_score = -1.0
      predicted_label = "Unknown"

      for ref_name, ref_vec in reference_database.items():
        ref_vec = ref_vec.to(device).view(-1)
        score = torch.nn.functional.cosine_similarity(query_vec, ref_vec, dim=0).item()

        if score > best_score:
          best_score = score
          predicted_label = ref_name

    y_true.append(class_name)
    y_pred.append(predicted_label)



if y_true:
  plt.figure(figsize=(12, 10))
  cm = confusion_matrix(y_true, y_pred, labels=classes)
  sns.heatmap(cm, annot=True, fmt='d', cmap="Blues", xticklabels=classes, yticklabels=classes)
  plt.xticks(rotation=90)
  plt.tight_layout()
  plt.show()
```

Figure 9: The inference loop: extracting embeddings for test images and calculating cosine similarity against all stored reference vectors to determine the best match.

3.8 Performance Evaluation

To visualize the performance of the one-shot learning model, a confusion matrix was generated using the Seaborn library, as shown in Figure 10. The rows represent the true classes, and the columns represent the predicted classes. The diagonal elements indicate the number of correct classifications for each class (out of 3 test samples per class).

As anticipated, the model demonstrates mixed performance, highlighting the inherent challenges of one-shot learning where the model relies on a single reference point to define an entire class.

- **Strengths:** Certain visually distinct landmarks, such as *Rumeli Hisarı*, *Sümela Manastırı*, and *Yerebatan Sarnıcı*, were classified perfectly (3/3 correct).
- **Weaknesses and Confusion:** Significant confusion occurred between structurally similar classes. For example, *Kız Kulesi* was frequently misclassified as *Galata Kulesi*, likely due to both being prominent tower structures. Similarly, mosques like *Ayasofya Camii* and *Sultan Ahmet Camii* showed varied performance due to shared architectural features (domes and minarets).

While a fully supervised training approach with a larger dataset would undoubtedly yield higher accuracy, these results successfully demonstrate the feasibility of the one-shot approach for rapid, low-data landmark identification, serving as a proof-of-concept for the multimodal guide.

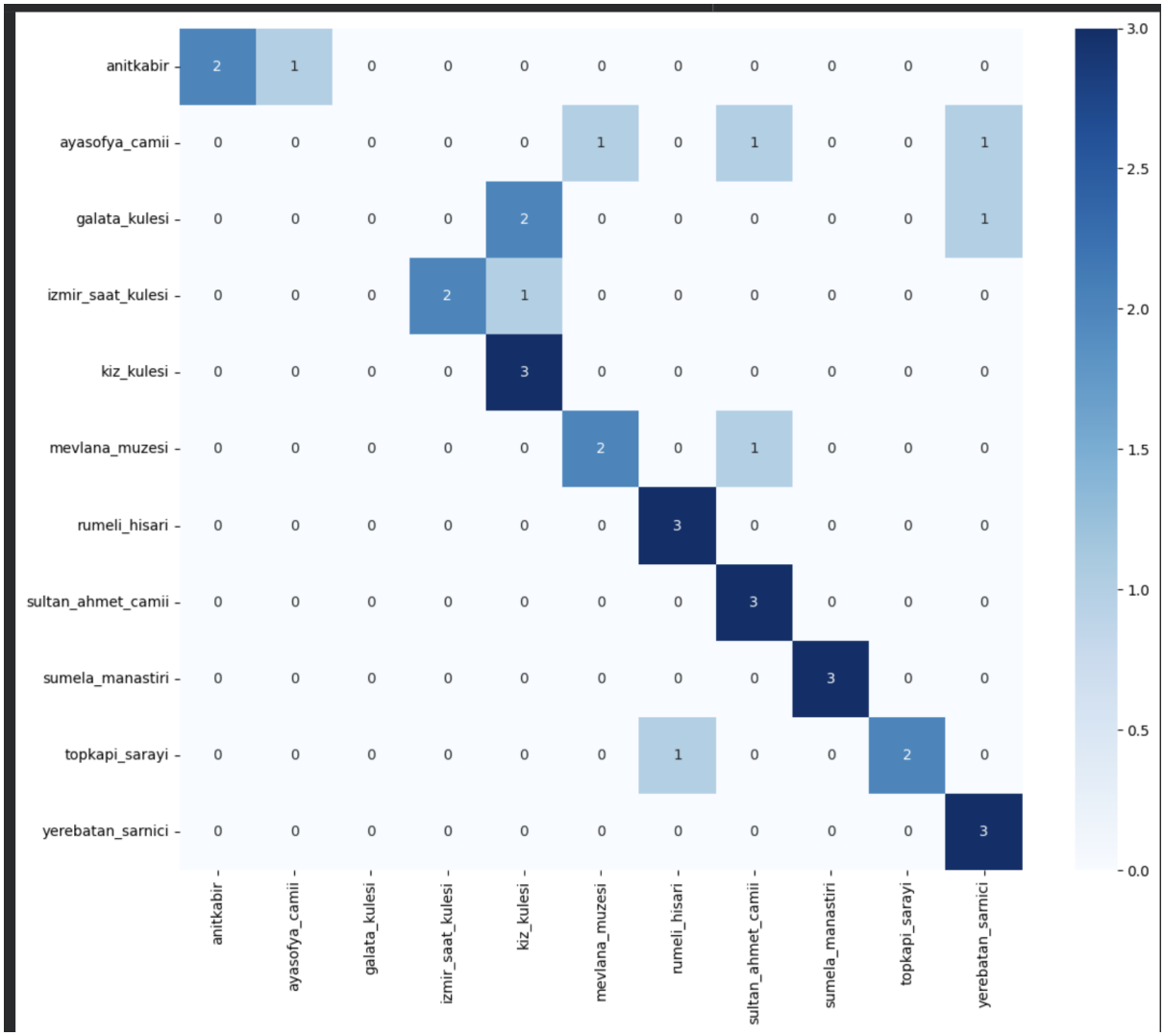


Figure 10: Confusion Matrix showing the classification results on the test set (3 images per class). Diagonal cells represent correct predictions.

3.9 RAG Pipeline Initialization

Following the visual classification, the system initializes the textual component of the multimodal guide. As shown in Figure 11, this phase involves setting up the Retrieval-Augmented Generation (RAG) pipeline to fetch and process historical information.

3.9.1 Knowledge Base Loading

The source of historical context is a pre-constructed JSON file, `turkish_landmarks_knowledge_base.js` which contains detailed descriptions of the landmarks scraped from Wikipedia.

The code reads this JSON file and restructures it into a Python dictionary ('knowledge$_db$').*Byiteratin*

3.9.2 Summarization Model

To convert the raw Wikipedia text into a concise narrative suitable for audio output, a summarization pipeline is established using the **Hugging Face Transformers** library.

The model selected for this task is `facebook/mbart-large-50-many-to-many-mmt`. This model was initialized within a 'summarization' pipeline and moved to the GPU ('device=device') to accelerate the text generation process.

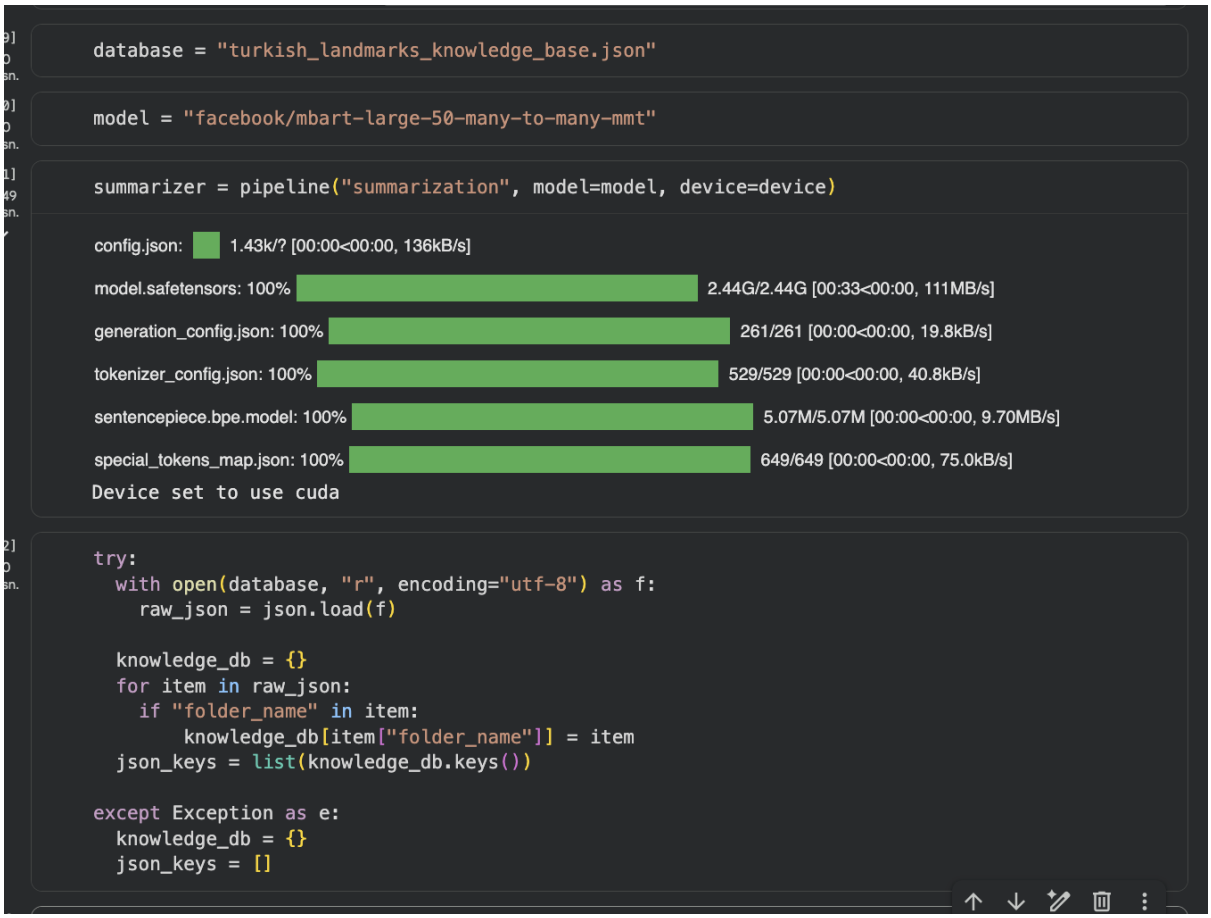


Figure 11: Initializing the RAG system: Loading the JSON knowledge base and setting up the MBART model for text summarization.

3.10 Knowledge Retrieval and Fuzzy Matching

To seamlessly connect the visual prediction with the text database, a robust retrieval mechanism was implemented, as shown in Figure 12.

3.10.1 Normalization and Matching

Since the class labels (folder names) might differ slightly from the keys in the JSON database (e.g., formatting differences), a 'normalize$_name$' $function is used to standardize strings by conver$

The core retrieval function, 'get$_landmark_info$', $employs **fuzzy matching** using Python's 'difflib.$

3.10.2 Prediction Encapsulation

The 'predict$_landmark$' $function encapsulates the entire inference pipeline described in the previous section$ $matching label along with the confidence score.$

```python
def normalize_name(name):
  return name.lower().replace("_", " ").strip()

def get_landmark_info(predicted_label, knowledge_db, json_keys):
  clean_pred = normalize_name(predicted_label)
  clean_keys = [normalize_name(k) for k in json_keys]

  matches = get_close_matches(clean_pred, clean_keys, n=1, cutoff=0.5)

  if matches:
    match_index = clean_keys.index(matches[0])
    real_key = json_keys[match_index]
    return knowledge_db[real_key]
  return None

def predict_landmark(image_path, model, processor, reference_db):
  image = Image.open(image_path).convert("RGB")
  inputs = processor(images=image, return_tensors="pt").to(device)

  with torch.no_grad():
    outputs = model(**inputs)

  query_vec = outputs.last_hidden_state[:, 0, :].view(-1)

  best_score = -1.0
  predicted_label = "Unknown"

  for ref_name, ref_vec in reference_db.items():
    ref_vec = ref_vec.to(device).view(-1)
    score = torch.nn.functional.cosine_similarity(query_vec, ref_vec, dim=0).item()

    if score > best_score:
      best_score = score
      predicted_label = ref_name

  return image, predicted_label, best_score
```

Figure 12: Implementation of the normalization utility, fuzzy matching for robust database retrieval, and the encapsulated prediction function.

3.11 Multimodal Output Generation

The final stage of the pipeline converts the retrieved historical data into a user-friendly format. As shown in Figure 13, this involves two distinct functions: text summarization and audio synthesis.

3.11.1 Context-Aware Summarization

The ‘generate_summary‘ function acts as the bridge between raw encyclopedic text and the user. It is designed to handle both English and Turkish inputs (‘lang=’en‘‘ or ‘’tr‘‘).

- **Input Truncation:** To ensure the text fits within the context window of the summarization model, the input is truncated to the first 2000 characters (‘input_text[:2000]‘).

- **Prompt Engineering:** A language-specific prompt prefix is prepended to the text (e.g., "Summarize this text in English: ") to guide the model's generation process.
- **Generation Parameters:** The summarization pipeline is called with specific constraints: 'max_length=150' to ensure brevity and 'min_length=40' to guarantee sufficient detail. The parameter 'do_sample=False' is used to make the output deterministic.

3.11.2 Text-to-Speech (TTS) Synthesis

The 'speak_text' function utilizes the **gTTS (Google Text-to-Speech)** library to vocalize the generated summary.

1. It accepts the summarized text and the target language code.
2. It generates an audio file named dynamically based on the language (e.g., 'summary_tr.mp3').
3. Finally, it uses 'IPython.display.Audio' with 'autoplay=True' to immediately play the audio narration within the Colab notebook environment.

```python
def generate_summary(info, summarizer_pipeline, lang='tr'):
  full_text = info.get('full_text', '') or info.get('summary', '')

  if lang == 'en':
    prompt_prefix = "Summarize this text in English: "
  else:
    prompt_prefix = "Aşağıdaki metni Türkçe özetle: "

  input_text = full_text[:2000]
  final_prompt = prompt_prefix + input_text


  summary_output = summarizer_pipeline(
      final_prompt,
      max_length=150,
      min_length=40,
      do_sample=False,
      truncation=True
  )
  return summary_output[0]['summary_text']
```

```python
def speak_text(text, lang='tr'):
    if not text or len(text) < 5:
      return

    filename = f"summary_{lang}.mp3"

    tts = gTTS(text=text, lang=lang, slow=False)
    tts.save(filename)
    display(Audio(filename, autoplay=True))
```

Figure 13: The multimodal generation functions: summarizing the retrieved text using a language-specific prompt and converting the result into an audio file using gTTS.

3.12 End-to-End Pipeline Execution

The final component of the project is the execution loop that integrates all previous modules into a cohesive user experience. As shown in Figure 14, the ‘run_full_pipeline‘ function orchestrates the entire process.

3.12.1 Orchestration Logic

The pipeline executes the following steps in sequence:

1. **Prediction:** It calls ‘predict_landmark‘ to classify the uploaded image and obtain the confidence score.
2. **Retrieval:** It fetches the corresponding historical context using ‘get_landmark_info‘.
3. **User Interaction:** It prompts the user to select their preferred language (TR/EN) via a standard input field.
4. **Generation:** It generates the summary and synthesizes the audio.
5. **Display:** Finally, it uses ‘display_results‘ to render the image with its predicted label and plays the generated audio.

3.12.2 Dynamic Interaction

To allow for real-time testing within the Google Colab environment, the ‘google.colab.files.upload()‘ method is utilized. This triggers a widget allowing the user to upload any image file from their local machine. The script immediately processes the uploaded file, demonstrating the system’s capability to handle novel, unseen data in a one-shot learning context.

```
def display_results(image, label, score, summary_text):
  plt.figure(figsize=(8, 5))
  plt.imshow(image)
  plt.axis('off')
  plt.title(f"{label}\nConf: {score:.2f}", fontsize=14, fontweight='bold', color='dark
  plt.show()
```

```
def run_full_pipeline(image_path):
  img, label, score = predict_landmark(image_path, model, processor, reference_databas

  info = get_landmark_info(label, knowledge_db, json_keys)

  lang_input = input("Language (TR/EN)? ").strip().lower()
  selected_lang = 'en' if lang_input in ['en', 'english', 'ing'] else 'tr'

  summary_text = generate_summary(info, pipeline, lang=selected_lang)

  clear_output(wait=True)
  display_results(img, label, score, summary_text)
  speak_text(summary_text, lang=selected_lang)


uploaded = files.upload()
for filename in uploaded.keys():
  run_full_pipeline(filename)
```

Figure 14: The full execution pipeline: orchestrating prediction, data retrieval, user input, and result display, triggered by a file upload widget.

3.13 Sample Execution and Results

Figure 15 illustrates a successful end-to-end execution of the HistoryLens pipeline.

1. **Input:** The user executes the cell and uses the upload widget to select a test image (e.g., `000004.jpg`).
2. **Processing:** The system confirms the use of the GPU (`Device set to use cuda:0`) for rapid inference.
3. **Visual Output:** The uploaded image is displayed with the predicted class label, **"yerebatan_sarnici"**, shown prominently as the title.
4. **Audio Output:** Following the identification, the system successfully retrieves the historical context, summarizes it, and generates an audio file. As seen at the bottom of the figure, an interactive audio player is embedded, allowing the user to listen to the generated narration immediately (duration: 0:59).

This result confirms that the one-shot learning model correctly matched the input image to the reference database and that the multimodal RAG pipeline functioned as intended.

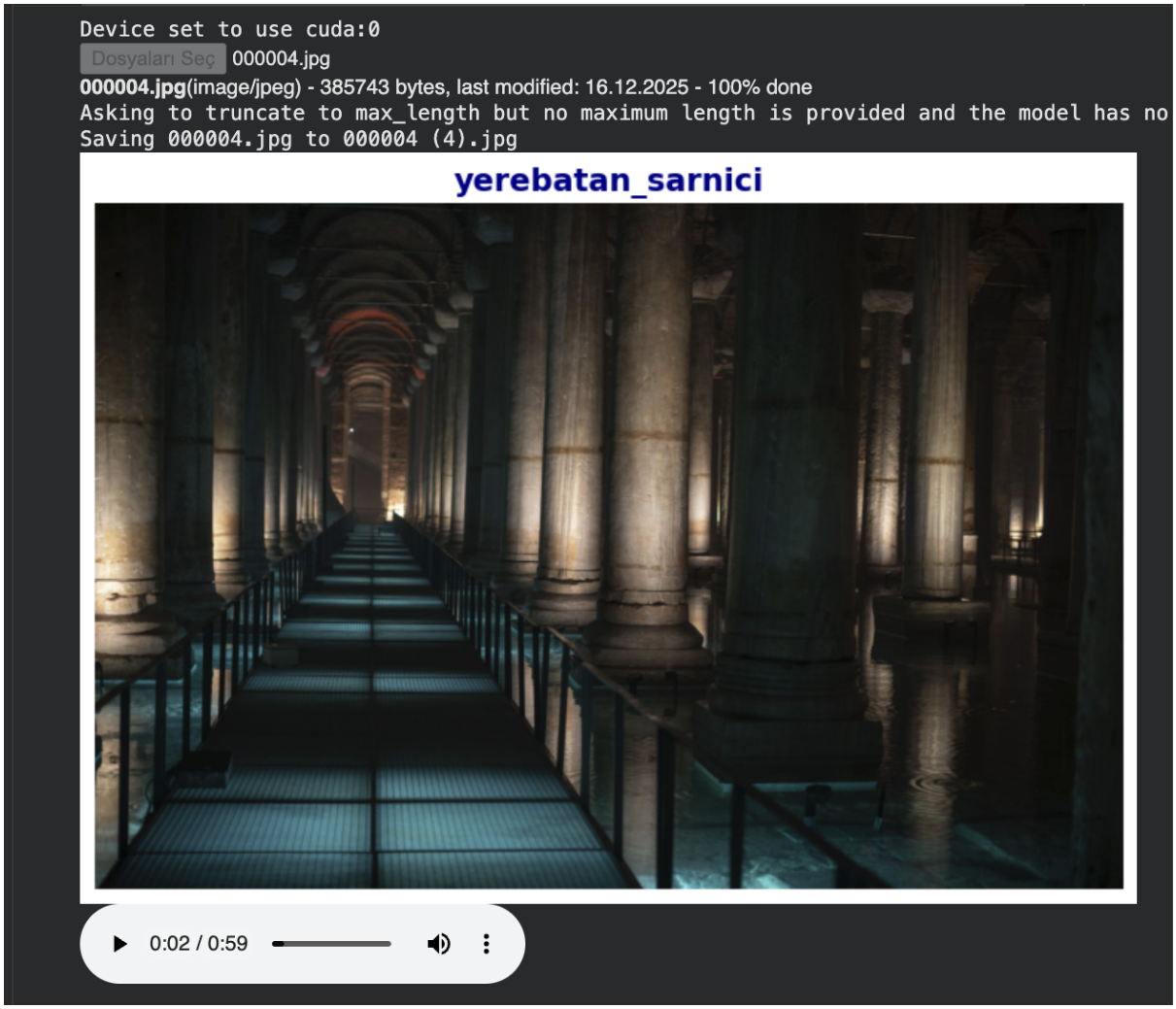


Figure 15: Final output of the system: The model correctly identifies 'Yerebatan Sarnıcı' and provides a playable audio summary in the selected language.

4 Conclusion

The **HistoryLens** project successfully demonstrates the potential of combining Computer Vision and Large Language Models to create an interactive, multimodal educational tool. By leveraging a Vision Transformer (ViT) in a **one-shot learning** configuration, the system was able to identify iconic Turkish landmarks using only a single reference image per class.

While the confusion matrix revealed certain limitations—specifically regarding the misclassification of structurally similar buildings like *Kız Kulesi* and *Galata Kulesi*—the project validates the feasibility of using embedding-based similarity for rapid, low-data classification tasks. The integration of a RAG pipeline further enriched the user experience, transforming simple image recognition into a comprehensive historical guide.

Future improvements could focus on expanding the dataset to include more diverse angles for each landmark, implementing advanced data augmentation techniques to improve robustness, and deploying the model as a mobile application to provide real-time assistance to tourists on-site.

Access to Code

The complete source code for this project, including the dataset download, model initialization, and RAG pipeline, is available via the following Google Colab link:

`https://colab.research.google.com/drive/1ebHBtREC9AEer9ZQdJZXAGd1ElzKED_9?usp=sharing`