Image Classification AppEngine Application

Faryal Tarique
Department of Computer Science
University of Porto
Porto, Portugal

Mariana Lourenço
Department of Computer Science
University of Porto
Porto, Portugal

Tomás Carmo
Department of Computer Science
University of Porto
Porto, Portugal

***Abstract*—This paper details the creation of an image classification application, deployed on Google AppEngine, that utilizes TensorFlow Lite models trained with Vertex AI. The system classifies images of furniture and home accessories, integrating BigQuery for dataset structuring and Cloud Vision API for enhanced label detection. The paper highlights the application's infrastructure, including a custom Docker image for versatile deployment across Cloud Shell and Cloud Run, and discusses the deployment of a cloud function for HTTP-triggered image classification. Furthermore, Firestore is used to log classification outcomes, accessible through a dedicated endpoint. The culmination of these technologies demonstrates a comprehensive approach to developing a cloud-based image classification system.**

I. Introduction

This project presents the development of a specialized cloud-based image classification application under Google Cloud (Project ID: "proj8824"). Hosted on AppEngine and accessible at [https://proj8824.uc.r.appspot.com/], the application is designed to categorize images of furniture and home accessories typically found in a home. Built upon Google AppEngine's robust infrastructure, it integrates a TensorFlow Lite model, trained using Vertex AI, to perform image classifications. This report outlines the process of constructing a BigQuery dataset from provided CSV files and details the programming of the app's endpoints, each of which queries the BigQuery data and references images stored in a public Cloud Storage bucket. Furthermore, the application utilizes the Cloud Vision API for label detection, offering a complementary classification mechanism to the TensorFlow Lite model.

In addition to the primary objectives, the project undertakes the task of defining a custom Docker image to encapsulate the app, enabling it to run in both Cloud Shell and Cloud Run environments. This report describes the specific commands used in the DockerFile and the rationale behind them, also indicating the URL of the successfully deployed container. A cloud function has been implemented to further augment the application's capabilities, allowing for HTTP-triggered image classification. Additionally, Firestore integration records the classification results, which are retrievable via a dedicated endpoint. This document explains the technical implementations of these enhancements and their contributions to the project's goals.

II. BigQuery Dataset Definition

A BigQuery dataset is defined from the provided CSV files and subsequently utilized in the application. The primary steps are outlined below, along with references to code snippets and explanations for clarity.

A. Project and Authentication Setup

For the project setup and authentication, the Google Cloud environment was initialized with a designated project ID $proj8824$, which served as a unique identifier for all subsequent operations. Authentication was conducted through google.colab, which facilitated secure access to BigQuery services, allowing the script to interact with and manage the project's cloud resources effectively (Fig. 1).

B. BigQuery Dataset Creation

Within BigQuery, a new dataset titled $openimages$ was instantiated to systematize the tables derived from the provided CSV datasets. This foundational step was crucial for structuring the subsequent data storage and retrieval processes within the cloud environment.

C. Defining Table Schemas

To translate the CSV files into a structured format within BigQuery, table schemas were defined (Fig. 2). Each table schema corresponds to the structure of the CSV data, which dictates the nature of the fields and the type of data they hold.

- **Overview of CSV Tables:**
  - *classes.csv:* Contains the label-to-name mappings for classifying images.
  - *image-labels.csv:* Holds the label associations for each image, tagging them with identifiable classes.
  - *relations.csv:* Captures complex relationships within images where two labels exist, clarifying the context of their link.

Here is a detailed description of the table schemas as visualized in the provided CSV files and data model diagram:

- **Classes Table Schema:** This schema provides the structure for the classes table, defining each entry as a unique image category or class. It includes:
  - *Label:* The unique identifier for a class, aligned with the 'Label' field in the *classes.csv* file.
  - *Description:* The textual description of the class, offering additional details about the label.
- **Image Labels Table Schema:** This schema outlines the *image_labels* table, linking images to their assigned classes. It comprises:

  - *ImageId:* The unique identifier for each image, corresponding to the file name in Cloud Storage.
  - *Label:* The class label associated with an image, denoting the category it falls under.
- **Relations Table Schema:** This schema dictates the composition of the relations table, which identifies relationships between labels within an image. It includes:
  - *ImageId:* Indicates the specific image in which the labels are related, similar to the *ImageId* in the *image_labels* schema.
  - *Label1* and *Label2:* The pair of labels involved in a relationship.
  - *Relation:* Describes the type of relationship between *Label1* and *Label2*.

D. Creating Tables

In the phase of table creation, the predefined schemas were utilized to construct the corresponding tables within the BigQuery openimages dataset (Fig. 2). This involved invoking BigQuery's table creation commands with the specified schemas to ensure that the resulting tables accurately reflected the structure necessary to accommodate the data from the CSV files. The process not only set up the data architecture but also verified the compatibility of the data types and required constraints, paving the way for a seamless data import and subsequent querying.

E. Loading Data into BigQuery

The data loading methodology involved using pandas DataFrames to ingest CSV files, which facilitated data manipulation and preparation (Fig. 3). The BigQuery client library was then employed to transfer the data from these DataFrames into the previously created BigQuery tables, thus populating the dataset with the necessary information for the application's functionality.

F. Validating Data Loading

To ensure the integrity of the data transfer, a verification process was executed post-upload (Fig. 4). This step entailed a systematic comparison of the row counts between the source pandas DataFrames and the destination BigQuery tables. Matching row counts verified a successful data upload, confirming that the full dataset was correctly and completely transferred into BigQuery without any data loss or corruption.

G. Sample Data Querying

Sample queries were designed to retrieve subsets of data from each table, providing a practical demonstration that the data was not only present but also query-able. This step was essential to confirm that the data was structured correctly and the tables were ready for use by the application, ensuring the data's integrity and the database's functionality.

H. Utilizing the BigQuery Dataset in the Application

For the application to leverage the constructed BigQuery dataset, it performs queries corresponding to the app's functional requirements. For instance, the application may execute a query on the $image_labels$ table to retrieve $ImageIds$ related to a certain class of musical instruments, enabling it to display images from the Cloud Storage bucket.

This process of dataset definition and querying forms the backbone of the app's data-driven functionality, allowing for dynamic interaction with the curated image dataset. The precise BigQuery queries used in the application's endpoints are detailed in Section 4, "Application Endpoints Implementation."

III. TensorFlow Lite Model Development

This project employs TensorFlow Lite (TF Lite) to develop an image classification model for furniture and home accessories, using Vertex AI. TF Lite is selected for its suitability for lightweight models, enabling fast on-device inference without the need for constant cloud connectivity. The model is designed to enhance the app by providing a quick and accurate classification of studio-related imagery. Here are the steps involved.

A. Data Preparation

Data preparation is the foundational step where images reflective of the home and decor theme are compiled. These images are sourced from the openimages dataset and subsequently stored in a dedicated Cloud Storage bucket for use in Vertex AI. As an additional step, the selected images are copied to another Google Cloud Storage (GCS) bucket which is used to prepare the CSV file for Vertex AI.

B. Preparing Vertex AI-Compatible CSV

For Vertex AI model training, a Python script was implemented to create a CSV file from the BigQuery dataset. This CSV file is structured to contain URLs from a Cloud Storage bucket and their associated labels. The process is broken down as follows:

- **BigQuery Client Setup:** Initializes access to BigQuery, setting up the connection to the specified project and dataset.
- **Table Definitions:** Identifies the relevant tables that contain the necessary image and label data.
- **Label Selection:** Selects labels specifically tied to home items, defining the scope of the model's classification capabilities (Fig. 5).
- **Label and Image Queries:** Extracts a predefined number of images per label, ensuring a diverse and ample dataset (Fig. 6). For each image, a URI is created that concatenates the base path from the Cloud Storage bucket $proj8824.appspot.com/ImageFiles/$ with the image file name and extension, providing a complete path for each image.

- **DataFrame Creation:** Organizes the data into a pandas DataFrame, sorting images and labels into designated subsets for training, validation, and test.
- **CSV File Generation:** Converts the DataFrame into a CSV file, preparing it for Vertex AI ingestion and subsequent model training (Fig. 8).

C. Creating Vertex AI Dataset and Importing the Data

The next phase involved creating a dataset within Vertex AI. The dataset was named and configured to the specific type of 'Image' for the purpose of 'Image classification (single-label)'. Consistency in region selection was maintained throughout, matching that of the Cloud Storage bucket used in the initial steps. With the dataset named and the objective set, the dataset was created. Following the creation, the new dataset, initially empty, was populated by importing the prepared data using the CSV file previously created, thereby finalizing the dataset's readiness for model training in Vertex AI.

D. Verifying the Images and Defining Splits

Utilizing the platform's 'Browse' feature, we confirmed that each image was correctly labeled, ensuring the integrity of the data for our ten selected classes: 'Chair', 'Desk', 'Bed', 'Bookcase', 'Door handle', 'Nightstand', 'Couch', 'Houseplant', 'Picture frame', and 'Lamp' (Fig. 9). This review process was critical as it assured us that the images to be used for model training were accurately associated with their corresponding labels.

By opting for a 'Custom data split', the dataset was divided into training, validation, and test subsets in the conventional ratios of 80%, 10%, and 10%, respectively (Fig. 10). Some challenges and errors encountered during the dataset preparation and image verification are detailed in Section 9.

E. Training and Evaluating the Model

The training of the TF Lite model was conducted using Vertex AI's AutoML, which automates the selection of the best model architecture to balance inference speed with accuracy. The model was trained on a dataset segmented into distinct categories like 'Lamp', 'Bookcase', and 'Door handle' to enable the precise classification of images within a home items context.

Upon completion of the training, the evaluation metrics, as observed in the Vertex AI platform, indicated an average precision of 0.687 across all labels, with 'Lamp' achieving the highest precision at 0.949. The recall metric stood at 45%, suggesting that while the model is quite precise in its predictions, it may not have captured all relevant instances across the dataset (Fig. 11).

Further insights were drawn from the confusion matrix, which provided a detailed breakdown of the model's classification accuracy per label (Fig. 12). High-performing labels like 'Bookcase' and 'Door handle' showcased strong precision, while 'Desk' and 'Chair' were identified as categories with room for improvement due to lower precision scores of 0.314 and 0.277, respectively.

F. Exporting Model to TF Lite Format

After the training and evaluation phases, the model was exported in the TensorFlow Lite (TF Lite) format suitable for deployment on mobile and edge devices. The export process also generated a dict.txt file, containing the labels associated with the model's output. This label file maps the model's numerical predictions back to understandable class names.

The model.tflite file, along with its metadata, was stored in a designated Cloud Storage bucket. To access the label information, the model.tflite file was unzipped, which extracted the dict.txt file containing all the image labels used during training (Fig. 13). This step finalized the model's readiness for integration into the application, allowing it to perform image classification tasks.

IV. Application Endpoints Implementation

In respect to the application endpoints, we implemented the following:

- **Image Info:** This endpoint receives an image ID as an argument and queries the BigQuery database for the respective image, displaying in the webpage not only the ID and image themselves, but also the classes associated with this picture. The query itself asks for the attribute "description" from an inner join with the tables "Image Labels" and "Classes" by "label", since we want to know the classes that a certain image has (similarly to the classes list query already built) (Fig. 14).
- **Relations:** The relations endpoint essentially lists all the different types of relations there can be between pictures and a respective count of these types. We achieved this by simply querying the database for the existing relations and aggregated these through relation to display their number (Fig. 15).
- **Image Search:** For the image search, we would receive a class description and an image result limit as arguments and displayed the pictures themselves and the given description. To get this result we queried the database for the Image IDs and description inner joining once again the tables "Image Labels" and "Classes" by "label" matching the given description and limiting result to the respective argument received (Fig. 16).
- **Relation Search:** Finally, we constructed the relation search endpoint, which received as arguments a relation, two classes bounded by that relation and a result limiter. It displayed the images respective to the classes with the relation in the middle to all relations that matched the arguments given. As for the query, we selected the image IDs and descriptions of the two images separately, and left inner joined the "Relations" table with the "Classes" table matching the first label, followed by a second left join, once more with the "Classes" table, matching the second label this time, where all should also match with both descriptions and relation arguments, limiting this result by the limiter chosen (Fig. 17).

On a final note, for each image ID we queried from the database we linked it with the respective image stored in

our cloud storage bucket when displaying it on the web application.

V. Containerization with Docker

For a versatile and efficient environment, we also defined a Docker image for our web application. Docker consists of a service that isolates an app making it available to different machines to run that same app without having to download the specific software carrying the project.

We start by creating our docker configuration file, or Dockerfile, where we included the application software version (python 3.9), we updated the software responsible for python packages (pip), copied to the docker environment the "requirements.txt" (containing a list of all packages used to be downloaded), installed or updated these, copied all the necessary project files, set a variable to send over the project id necessary for the application instance and exposed the port used by our python app finalizing this file with running that "main.py" app (Fig. 18).

After the creation of the Dockerfile, we build an image of the project by running "docker build . -t bdcc-app", associating the image with the tag "bdcc-app". Next, we enable the google cloud registry service to store the image to a docker registry, push the respective image and deploy it on Cloud Run. This service essentially allows us to use the docker container created in the Google Cloud Platform. The Cloud Run web app created can be seen in https://bdcc-app-ijssowhvkq-oa.a.run.app.

VI. Cloud Function for Image Classification

The implementation of a serverless image classification service utilizes Google Cloud Functions to provide a scalable, cost-efficient, and maintenance-free solution for classifying images. This service simplifies the deployment and management of machine learning models by leveraging the cloud's power, thus allowing for dynamic scaling in response to demand without the need for dedicated infrastructure.

A. How It Works:

- *Triggering the Cloud Function:* The process begins with an HTTP request trigger. The cloud function is designed to activate upon receiving a GET request that includes a URL parameter specifying the image's location for classification.
- *Processing the Request:* The cloud function executes the following steps upon activation:
  1) **Extracting the Image URL:** It parses the incoming HTTP request to extract the `url` parameter, which contains the address of the image to be classified.
  2) **Downloading the Image:** The function then downloads the image from the provided URL, preparing it for the classification process.
- *Image Classification:*
  1) **Loading the TensorFlow Lite Model:** A TensorFlow Lite model is preloaded into the function's runtime for immediate use upon invocation.
  2) **Preparing the Image:** The image is processed to match the input requirements of the TensorFlow Lite model.
  3) **Performing Classification:** The model classifies the image, returning predictions in the form of labels and confidence scores.
- *Responding to the Request:* The function formats the classification results into a JSON structure, which is sent back to the requester as the HTTP response. This includes detailed classification information such as identified object labels and their confidence levels.

B. Example Usage:

An example HTTP request to the cloud function is as follows:

https://display-img-func-ijssowhvkq-uc.a.run.app/?url=https://farm3.staticflickr.com/6205/6146854270_2325bbf401_o.jpg&min_confidence=0.01.

This request demonstrates how the cloud function is triggered and subsequently processes the image classification. The first argument consists of a valid image URL and the second one is the minimum confidence level of the model (like the web application, this second parameter is optional, using the value 0.25 as a default)

VII. Firestore Integration

The integration's goal was to reliably log the image classifications conducted by the TensorFlow Lite model into Firestore and to furnish an endpoint that retrieves and lists these results.

A. Implementation Overview

Firestore was integrated within the Flask application structure in *main.py*. This setup enabled the capture and storage of classification outcomes, effectively associating each image URL with its classification labels and a timestamp in Firestore.

B. Operational Workflow

- Upon each image classification via the *image_classify* endpoint, the TF Lite model's results are preserved in Firestore within the 'classifications' collection. Each record consists of the filename, the classification labels, the storage bucket name, and the timestamp of classification (Fig. 19).
- The stored documents in Firestore comprise detailed classification information and relevant metadata for the images.
- A specialized endpoint, *classification_results*, has been implemented to compile and display the historical classification data from Firestore to the users (Fig. 20).

This Firestore integration contributes to a robust data layer for the application, enabling persistent tracking of classification activities and enhancing the service offered to end-users.

VIII. Integration of Cloud Vision API

An application endpoint was developed to incorporate Google Cloud Vision API for robust image classification, enabling label detection with high accuracy.

A. Implementation Overview

The `VisionAPI.py` module houses the `detect_labels_uri(uri)` function, which is integral to the application's interaction with the Google Cloud Vision API. This function performs the following operations:

- Instantiates a client for the Vision API.
- Sets the source URI for the image to be analyzed.
- Executes the `label_detection` method to obtain labels from the image.
- Extracts and returns the labels and confidence scores from the response (Fig. 21).

B. Operational Workflow

- The endpoint receives an image URL and a minimum confidence threshold from the user.
- These inputs are passed to the `detect_labels_uri` function.
- The function communicates with the Google Cloud Vision API, submitting the image for label detection.
- The Vision API processes the image and returns a list of labels and confidence scores.
- The labels and scores are formatted for delivery through the application's user interface (Fig. 22).

IX. Additional Challenges and Innovations

Here are some of the additional challenges faced and the innovative solutions or features developed.

A. BigQuery Dataset and CSV Creation

The process of creating a BigQuery dataset and preparing a corresponding CSV file for Vertex AI presented unique challenges. Each of these challenges required thoughtful solutions to ensure the integrity and usability of the machine learning dataset.

- **Challenge 1: Ensuring Unique Images Across Labels**
  Problem: A significant challenge in dataset preparation is ensuring that each image is only used once across training, validation, and test sets. This is critical to prevent data leakage and ensure an accurate evaluation of the machine learning model. The challenge is compounded when an image is associated with multiple labels.
  Solution: To address this, we implemented a comprehensive checking mechanism that involved:
  - **Aggregating Image Data:** Crafting a query to aggregate images based on their labels, ensuring that each image was counted only once per label. This approach was crucial for identifying images with multiple labels.
  - **De-duplication Logic:** Implementing logic in the data preparation script to check if an image already exists in the dataset before adding it. If an image was associated with multiple labels, it was allocated to the first encountered label, ensuring that no image appeared in more than one label group (Fig. 6).
- **Challenge 2: Automated Image Copying to GCS Bucket**
  Problem: For training models on Vertex AI, images need to be stored in a Google Cloud Storage (GCS) bucket. Manually copying thousands of images from a public dataset to a private bucket is not only inefficient but also prone to errors.
  Solution: An automated script was developed to iterate over the selected images, copy them from the public `bdcc_open_images_dataset` bucket to the project's private bucket, and maintain the mapping of image IDs to their new GCS URIs (Fig. 7). This script utilized the Google Cloud Storage client library to efficiently handle the copy operation and log any encountered errors.
- **Challenge 3: Dealing with Memory Issues**
  Problem: Upon deploying our application image to the Cloud Run environment, the application wouldn't start due to insufficient memory space. There was not much to be optimized in the script since we needed all libraries.
  Solution: We edited and deployed a new revision on the Cloud Run container with 2GB memory to make sure it had enough to work efficiently.
- **Challenge 4: Permissions Problems**
  Problem: Another challenge related to the Cloud Run deployment was the fact that it wouldn't allow access to the webpage when opened, giving an unauthorized error.
  Solution: We added permissions to all Users, specifically the Cloud Run Invoker role, so that anyone could access the application.

X. Results and Discussion

The development and deployment of our image classification application on the Google's AppEngine involved the use of a TensorFlow Lite model trained via Vertex AI's AutoML. This approach optimized the balance between inference speed and accuracy, tailoring the model to effectively categorize home item images into distinct categories such as 'Lamp', 'Bookcase', and 'Door handle'. The evaluation metrics post-training presented an enlightening view of the model's capabilities. The model achieved an average precision of 0.687 across all labels, a noteworthy accomplishment with 'Lamp' category reaching the highest precision at 0.949. However, the recall metric at 45% indicated a limitation in the model's ability to identify all pertinent instances within the dataset, suggesting a potential area for refinement. A deeper analysis via the confusion matrix unveiled the model's classification efficacy per label. While 'Bookcase' and 'Door handle' categories demonstrated high precision, indicating a robust ability to identify these items accurately, the 'Desk' and 'Chair' categories emerged as areas needing enhancement due to their lower precision scores of 0.314 and 0.277, respectively.

XI. Conclusion

The project showcases a successful integration of cloud and AI technologies. The model demonstrates high precision in identifying items like 'Lamp' but reveals the challenges of consistent accuracy across varied categories. The

results—while impressive in some respects—indicate a need for further refinement, particularly to improve the recall rate.

Future work will focus on addressing these discrepancies by enriching the dataset, further tuning the model, and exploring advanced training techniques to enhance both precision and recall. The goal is to evolve our application into a more comprehensive and reliable tool for home item classification, paving the way for broader applications and innovations in the domain of image classification.

XII. Appendix

BigQuery Dataset Definition

Authentication

```
PROJECT_ID = 'proj8824'
```

```
[ ] from google.colab import auth
    auth.authenticate_user()
    !gcloud config set project {PROJECT_ID}
```

```
Updated property [core/project].
```

Fig. 1: Authenticating

```
DATASET_ID = "openimages"

# Dataset creation
dataset_ref = client.dataset(DATASET_ID)
dataset = bq.Dataset(dataset_ref)
dataset.location = "US"
client.create_dataset(dataset, exists_ok=True)

# Schema for Classes table
classes_schema = [
    bq.SchemaField("Label", "STRING", mode="REQUIRED"),
    bq.SchemaField("Description", "STRING", mode="REQUIRED"),
]

# Define schema for image_labels table
image_labels_schema = [
    bq.SchemaField("ImageId", "STRING", mode="REQUIRED"),
    bq.SchemaField("Label", "STRING", mode="REQUIRED"),
]

# Schema for relations table
relations_schema = [
    bq.SchemaField("ImageId", "STRING", mode="REQUIRED"),
    bq.SchemaField("Label1", "STRING", mode="REQUIRED"),
    bq.SchemaField("Relation", "STRING", mode="REQUIRED"),
    bq.SchemaField("Label2", "STRING", mode="REQUIRED"),
]

# Table creation
tables = {
    'classes': classes_schema,
    'image_labels': image_labels_schema,
    'relations': relations_schema
}

for table_id, schema in tables.items():
    table_ref = dataset_ref.table(table_id)
    table = bq.Table(table_ref, schema=schema)
    client.create_table(table, exists_ok=True)

print("BigQuery dataset and tables created successfully.")
```

Fig. 2: Creating BQ Tables

```
[ ] import pandas as pd

    image_labels_df = pd.read_csv('image-labels.csv')
    classes_df = pd.read_csv('classes.csv')
    relations_df = pd.read_csv('relations.csv')
```

```
[ ] # Loading data from DataFrame to BigQuery table
    def load_data_to_bigquery(df, table_id):
        job_config = bq.LoadJobConfig(autodetect=True)  # Let BigQuery determine the schema automatically
        job = client.load_table_from_dataframe(df, table_id, job_config=job_config)
        job.result()  # Wait for the job to complete

    # Loading the 'classes' data
    classes_table_id = f"{PROJECT_ID}.openimages.classes"
    load_data_to_bigquery(classes_df, classes_table_id)

    # Loading the 'image_labels' data
    image_labels_table_id = f"{PROJECT_ID}.openimages.image_labels"
    load_data_to_bigquery(image_labels_df, image_labels_table_id)

    # Loading the 'relations' data
    relations_table_id = f"{PROJECT_ID}.openimages.relations"
    load_data_to_bigquery(relations_df, relations_table_id)

    print("Data has been uploaded to BigQuery.")
```

Fig. 3: Loading Data Into BigQuery

```
# Comparing the row counts for each table
for table_name, dataframe in [('classes', classes_df), ('image_labels', image_labels_df), ('relations', relations_df)]:
    table_id = f"{PROJECT_ID}.{DATASET_ID}.{table_name}"
    table = client.get_table(table_id)  # Make an API request.
    print(f"The table {table_id} contains {table.num_rows} rows.")

    expected_row_count = len(dataframe.index)
    print(f"The DataFrame for {table_name} contains {expected_row_count} rows.")

    if expected_row_count == table.num_rows:
        print(f"Row count matches for {table_id}!")
    else:
        print(f"Row count mismatch for {table_id}!")
```

Fig. 4: Validating Data Loading

TensorFlow Lite Model Development

```
PROJECT_ID = 'proj8824'
BQ_CLIENT = bigquery.Client(project=PROJECT_ID)
STORAGE_CLIENT = storage.Client(project=PROJECT_ID)


SOURCE_BUCKET_NAME = 'bdcc_open_images_dataset'
TARGET_BUCKET_NAME = 'proj8824.appspot.com'


DATASET_ID = 'openimages'
IMAGES_TABLE_NAME = 'image_labels'
LABELS_TABLE_NAME = 'classes'

# Specifying labels to include
INCLUDE_LABEL_DESCRIPTIONS = [
    'Chair', 'Desk', 'Bed', 'Bookcase', 'Door handle', 'Nightstand', 'Couch', 'Houseplant', 'Picture frame', 'Lamp'
]
```

Fig. 5: Specifying Labels to Include

```
# Query to fetch images, considering additional non-specified labels are acceptable
query = f"""
SELECT IL.ImageId, CL.Description
FROM `{PROJECT_ID}.{DATASET_ID}.{IMAGES_TABLE_NAME}` AS IL
JOIN `{PROJECT_ID}.{DATASET_ID}.{LABELS_TABLE_NAME}` AS CL ON IL.Label = CL.Label
WHERE CL.Description IN UNNEST(@include_labels)
GROUP BY IL.ImageId, CL.Description
"""

job_config = bigquery.QueryJobConfig(
    query_parameters=[
        bigquery.ArrayQueryParameter("include_labels", "STRING", INCLUDE_LABEL_DESCRIPTIONS)
    ]
)

query_job = BQ_CLIENT.query(query, job_config=job_config)
results = query_job.result()

# Organizing images by their label, ensuring uniqueness across categories
used_images = set()
images_by_label = {label: [] for label in INCLUDE_LABEL_DESCRIPTIONS}

for row in results:
    image_id = row["ImageId"]
    label = row["Description"]
    if image_id not in used_images and len(images_by_label[label]) < 100:
        images_by_label[label].append(image_id)
        used_images.add(image_id)  # Mark this image as used
```

Fig. 6: Query to Fetch Unique Images

```
# Copying an image from the source to the target bucket and returning the target URI
def copy_image_to_target_bucket(source_image_id):
    source_blob_path = f"images/{source_image_id}.jpg"
    target_blob_path = f"images/{source_image_id}.jpg"
    source_blob = STORAGE_CLIENT.bucket(SOURCE_BUCKET_NAME).blob(source_blob_path)
    target_blob = STORAGE_CLIENT.bucket(TARGET_BUCKET_NAME).copy_blob(source_blob, STORAGE_CLIENT.bucket(TARGET_BUCKET_NAME), target_blob_path)
    return f"gs://{TARGET_BUCKET_NAME}/{target_blob_path}"
```

Fig. 7: Copying Selected Image Files to GCS

```
# Prepareing the CSV data
csv_data = []

for label, images in images_by_label.items():
    for idx, image_id in enumerate(images):
        # Determining the dataset split (training, validation, test) based on the index
        if idx < 80:
            data_type = 'training'
        elif idx < 90:
            data_type = 'validation'
        else:
            data_type = 'test'

        # Copying the image to our bucket and getting the target URI
        target_image_uri = copy_image_to_target_bucket(image_id)

        # Appending the data to the CSV list
        csv_data.append({
            "set": data_type,
            "image_uri": target_image_uri,
            "label": label
        })

# Creating a dataframe
df = pd.DataFrame(csv_data)

# Saving the DataFrame to a CSV file
csv_file_path = "vertex_ai_dataset.csv"
df.to_csv(csv_file_path, index=False)
```

Fig. 8: Creating the Vertex AI CSV File

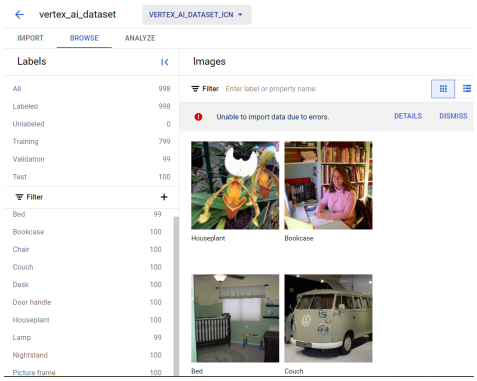

Fig. 9: Vertex AI's Browse Feature

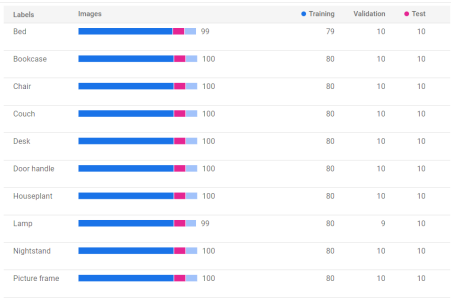

Fig. 10: Custom Data Split

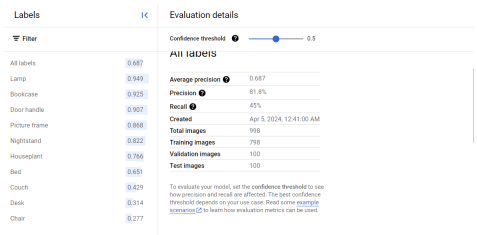

Fig. 11: Model Evaluation Result

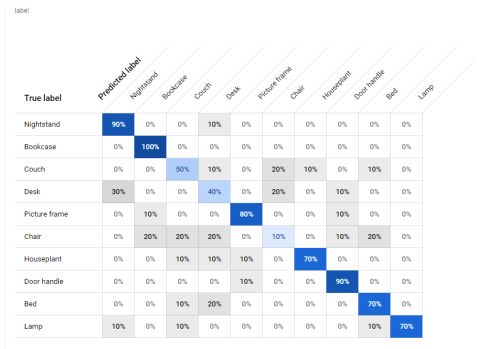

Fig. 12: Confusion Matrix for the AI Model

```
tomas_alportel@cloudshell:~/bdcc/project/app (proj8824)$ cat static/tflite/dict.txt
Nightstand
Bookcase
Couch
Desk
Picture frame
Chair
Houseplant
Door handle
Bed
Lamp
```

Fig. 13: Image Labels in dict.txt file

Application Endpoints Implementation

```python
@app.route('/image_info')
def image_info():
    image_id = flask.request.args.get('image_id')

    results = BQ_CLIENT.query(
    '''
        Select Description
        FROM `proj8824.openimages.image_labels`
        JOIN `proj8824.openimages.classes` USING(Label)
        WHERE ImageId = '{}'
    '''.format(image_id)).result()

    return flask.render_template('image_info.html', results=results, image_id=image_id)
```

Fig. 14: Image Info Endpoint

```python
@app.route('/relations')
def relations():
    results = BQ_CLIENT.query(
    '''
        Select relation, count(*)
        From openimages.relations
        Group By relation
    ''').result()

    return flask.render_template('relations.html', results=results)
```

Fig. 15: Relation Types Endpoint

```python
@app.route('/image_search')
def image_search():
    description = flask.request.args.get('description', default='')
    image_limit = flask.request.args.get('image_limit', default=10, type=int)

    results = BQ_CLIENT.query(
    '''
        Select ImageId
        FROM `proj8824.openimages.image_labels`
        JOIN `proj8824.openimages.classes` USING(Label)
        WHERE Description = '{}'
        LIMIT {}
    '''.format(image_limit)).result()
    return flask.render_template('image_search.html', results=results, description=description)
```

Fig. 16: Image Search Endpoint

```python
@app.route('/relation_search')
def relation_search():
    class1 = flask.request.args.get('class1', default='%')
    relation = flask.request.args.get('relation', default='%')
    class2 = flask.request.args.get('class2', default='%')
    image_limit = flask.request.args.get('image_limit', default=10, type=int)

    results = BQ_CLIENT.query(
    '''
        Select a.ImageId, b.description as Desc1, a.relation, c.description as Desc2
        From openimages.relations as a
        Left Join openimages.classes as b On a.label1=b.label
        Left Join openimages.classes as c On a.label2=c.label
        Where b.description LIKE '{}' AND a.relation LIKE '{}' AND c.description LIKE '{}'
        Limit {}
    '''.format(class1,relation,class2,image_limit)).result()

    return flask.render_template('relation_search.html', results=results)
```

Fig. 17: Relation Search Endpoint

Docker

home > tomas_alportel > bdcc > project > app > Dockerfile

```
# our base image
FROM python:3.9
# Install python and pip
# RUN apk add --update py2-pip
# upgrade pip
RUN pip install --upgrade pip
# install Python modules needed by the Python app
COPY requirements.txt /usr/src/app/
RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt
# copy files required for the app to run
COPY main.py /usr/src/app/
COPY templates/* /usr/src/app/templates/
COPY static/* /usr/src/app/static/
COPY static/tflite/* /usr/src/app/static/tflite/
COPY tfmodel.py /usr/src/app/
COPY keys/* /usr/src/app/keys/
COPY VisionAPI.py /usr/src/app/
# Extra
ENV GOOGLE_CLOUD_PROJECT=proj8824
# tell the port number the container should expose
EXPOSE 8080
# run the application
CMD ["python", "/usr/src/app/main.py"]
```

Fig. 18: Dockerfile

Firestore and Cloud Vision API

```
# Create a document in Firestore
doc_ref = db.collection('classifications').document()
doc_ref.set({
    'filename': file.filename,
    'classifications': classifications,
    'bucket_name': APP_BUCKET.name,
    'timestamp': firestore.SERVER_TIMESTAMP
})
```

Fig. 19: Firestore app route

```
# Initialize Firestore DB

@app.route('/classification_results')
def classification_results():
    docs = db.collection
    ('classifications').order_by('timestamp', direction=firestore.Query.DESCENDING).stream()

    classifications = [doc.to_dict() for doc in docs]

    return flask.render_template('classification_results.html', classifications=classifications)
```

Fig. 20: Firestore app route

```
def detect_labels_uri(uri):
    """Detects labels in the file located in Google Cloud Storage or on the
    Web."""
    from google.cloud import vision

    client = vision.ImageAnnotatorClient()
    image = vision.Image()
    image.source.image_uri = uri

    response = client.label_detection(image=image)
    labels = response.label_annotations

    if response.error.message:
        raise Exception(
            "{}\nFor more info on error messages, check: "
            "https://cloud.google.com/apis/design/errors".format(response.error.message)
        )

    return labels
```

Fig. 21: Detect labels uri

```
@app.route('/CloudVision', methods=['POST'])
def CloudVision():
    image_url = flask.request.form['image_url']
    min_confidence = float(flask.request.form['min_confidence']) / 100  # Assuming input is 0-100, converting to 0-1

    labels = detect_labels_uri(image_url)

    # Validate URL format and image extension
    if not re.match(r'^https?:\/\/.*\.(jpg|jpeg|png|gif)$', image_url, re.IGNORECASE):
        flash('Please enter a valid image URL ending with .jpg, .jpeg, .png, or .gif.', 'warning')
        return redirect(url_for('index'))  # Assuming 'index' is the name of your route for the form

    # Process the labels to fit the expected structure for CloudVision.html
    processed_labels = []
    for label in labels:
        if label.score >= min_confidence:
            processed_labels.append({
                'description': label.description,
                'score': label.score
            })

    # Prepare data for the template
    data = {
        'results': [{
            'image_url': image_url,
            'labels': processed_labels
        }],
        'min_confidence': min_confidence * 100  # Convert back to percentage for display
    }
```

Fig. 22: Cloud Vision Route