BDCC - Project 1

2023/2024 · 2nd Semester · CC4093

Class: TP1

Daniela dos Santos Tomás	202004946
Diogo Filipe Faia Nunes	202007895
Diogo Pinheiro Almeida	202006059

Base Project

All application endpoints that were missing and their corresponding html pages are now fully implemented and working just like the example project provided.

BigQuery queries

Relations

The relations endpoint fetches distinct relations from the dataset along with their counts. The BigQuery query gets this information directly from the relations table, and the results are ordered alphabetically by relation.

```
SELECT DISTINCT Relation, COUNT(Relation)
FROM `bdcc-project1-417811.openimages.relations`
GROUP BY Relation
ORDER BY Relation
```

Image Info

The image info endpoint provides information about a specific image identified by the image ID submitted by the user. For this endpoint, we made two separate queries:

The first fetches the class descriptions associated with the image using the image ID.

```
SELECT Description
FROM `bdcc-project1-417811.openimages.image_labels`
    JOIN `bdcc-project1-417811.openimages.classes` USING(Label)
WHERE imageId = @imageId
ORDER BY Description
```

The other fetches the relationships for that image.

```
SELECT c1.Description, Relation, c2.Description
FROM bdcc-project1-417811.openimages.classes AS c1
    JOIN bdcc-project1-417811.openimages.relations ON c1.Label = Label1
    JOIN bdcc-project1-417811.openimages.classes AS c2 ON c2.Label = Label2
WHERE ImageId = @imageId
GROUP BY c1.Description, Relation, c2.Description
```

Image Search

The image search endpoint allows users to search for images based on a specific description. It fetches image IDs from the image_labels table based on the provided description and with a limit on the number of images specified by the image_limit parameter.

```
SELECT ImageId
FROM bdcc-project1-417811.openimages.classes
    JOIN bdcc-project1-417811.openimages.image_labels USING (Label)
WHERE Description = @description
GROUP BY ImageId
LIMIT @image_limit
```

Relation Search

The relation search endpoint searches for images based on specific class relations. The query joins relations, image_labels, and classes tables to fetch image IDs along along with descriptions of two related classes based on the provided parameters like class1, relation, and class2. The number of results is limited based on the image_limit. `@class1 <> "" AND @class2 <> ""` ensures that empty string class descriptions are avoided for both classes.

```
SELECT ImageId, c1.Description, c2.Description
FROM bdcc-project1-417811.openimages.relations
    JOIN bdcc-project1-417811.openimages.image_labels USING (ImageId)
    JOIN bdcc-project1-417811.openimages.classes c1
        ON c1.Label = Label1 AND c1.Description LIKE CONCAT('%', @class1, '%')
    JOIN bdcc-project1-417811.openimages.classes c2
        ON c2.Label = Label2 AND c2.Description LIKE CONCAT('%', @class2, '%')
WHERE Relation = @relation AND @class1 <> "" AND @class2 <> ""
GROUP BY ImageId, c1.Description, c2.Description
ORDER BY ImageId, c1.Description, c2.Description
LIMIT @image_limit
```

Finally, all of the endpoints render the template with the obtained data.

TensorFlow data set

Our TensorFlow data set preparation notebook works by retrieving the ids of 100 images from each of the ten classes we select, then copying the image from the open_images bucket to our bucket so we can train our vertex model. At the same time, it writes the link to each one and tags it with the class it belongs to and whether it's for training, validation, or testing (80/10/10 split).

Docker Image

After a few rounds of troubleshooting, we discovered that when we hosted the docker image with the code specifying "127.0.0.1" (loopback address), the application would reject connections from any machine other than itself (which makes sense), so we changed the host address to "0.0.0.0", and everything began to work as expected. This was the only code change required to convert the application into a working Docker image.

```
FROM python:3.9-slim-buster

# Set the working directory in the container
```

```
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

#Make sure pip package is up-to-date
RUN pip install --upgrade pip

# Install any needed dependencies specified in requirements.txt
RUN pip install -r requirements.txt

#Establish the project ID. (this is essential to run on local development and
testing environments, which are not on Google Cloud)
#### NOTE #### replace below with your project id
ENV GOOGLE_CLOUD_PROJECT="<google cloud project id>"

# Expose the port Flask will run on(must be the same as defined in the "app.run()"
command in the end of the main.py file)
EXPOSE 8080

# Define the command to run the Flask application
CMD ["python3", "main.py"]
```

Finally, if you wish to upload the container image and start an instance, simply perform these commands while in the project's root directory.

```
sudo docker build . -t <project_name>

docker run --rm -it -p 8080:8080 <project_name>

docker tag <project_name> gcr.io/<google cloud project id>/<project_name>

docker push gcr.io/<google cloud project id>/<project_name>

#In our experience, the application did not run with the default 512Mi amount,
therefore we changed it to the next available choice.

gcloud run deploy <project_name> --image gcr.io/<google cloud project
id>/<project_name> --memory 1024Mi
```

Finally, the `docker run` command will not work unless you create a service account key in json format and specify the path to it in the command

```
ENV GOOGLE_APPLICATION_CREDENTIALS="<path/to/key.json>"
```

on the dockerfile.

You may visit our deployment here