Professor Eduardo Marques

Project 1: Big Data and Cloud Computing

Google Cloud Project ID: “bdcc24-project1”

AppEngine URL: bdcc24-project1.oa.r.appspot.com

April, 2024

Pedro Leite - 201906697

José Rodrigues - 202006455

Table of Contents

1. Introduction

In this project, our objective is to develop an AppEngine application that uses the images from Open Images dataset, providing users with insightful information about images through interfaces. Leveraging Google Cloud's tools like BigQuery and Cloud Storage, alongside TensorFlow Lite models trained using Vertex AI, the app aims to deliver efficient image classification and analysis capabilities. A functional TF Lite model and a partially finished application are given as a starting point, and a demo application with all the basic features, to help with the evaluation. This paper describes every step, towards our final goal.

2. BigQuery

2.1. Bucket and Dataset

We've started by creating the bucket "bucket-bdcc24-project1", where we uploaded the CSV files "classes.csv", "image-labels.csv" and "relations.csv", supplied. In the BigQuery Studio, we created the dataset "openimages". Where we defined the tables, "classes", "image-labels" and "relations", using the CSV files from the bucket, previously created.

2.2. Project ID, Authentication and BigQuery Client

Then we created a python notebook, "notebook-bdcc-project1.ipynb", in order to define the new tables for our dataset. We've started by setting the project id, google authentication, for a colab environment and the BigQuery client, to interact with the previously created dataset tables.

```
#Project
PROJECT_ID = 'bdcc24-project1' #@param {type: "string"}

#Authentication
from google.colab import auth
auth.authenticate_user()
!gcloud config set project {PROJECT_ID}

#Big Query Client
import google.cloud.bigquery as bq
client = bq.Client(project=PROJECT_ID)
```

2.3. Convert Tables to Pandas Dataframe

We've used SQL queries to select the tables, and then we converted them to a Pandas dataframe. We had to drop the first rows and rename the columns, because the name of the columns was the type of field, while the actual label, was in the first row.

```
#Classes
classes_ref = dataset_id+".classes"
classes_df = client.query(f"SELECT * FROM `{classes_ref}`").to_dataframe()
classes_df = classes_df.rename(columns={"string_field_0": "Label",
                                        "string_field_1": "Description"})
classes_df = classes_df.drop(0)
```

2.4. Load Dataframes to BigQuery

We created the new tables (names ending in "2"), defined the schemas and then loaded them, into our dataset. And now we have the BigQuery tables, for the "classes", "image-labels" and "relations", with the correct formation.

```
#Classes
classes_ref2 = dataset_id+".classes2"
client.delete_table(classes_ref2, not_found_ok=True)
classes_table = bq.Table(classes_ref2)
classes_table.schema = (
        bq.SchemaField('Label',       'STRING'),
        bq.SchemaField('Description','STRING')
)
client.create_table(classes_table)
```

2.5. Create Joined Pandas Dataframes

We've then defined 3 new dataframes, using Pandas. The first one, "joined", is the merge between "classes" and "image-labels", on "Label". This dataframe contains, every "ImageId" and it's corresponding "Label" and "Description".

```
#Joined1
joined_df = pd.merge(classes_df, image_labels_df, on='Label')
```

The second one, "joined2", is the merge between "relations" and "classes". To do this we had to rename the columns "Label" and "Description", from "classes", to "Label1" and "Description1", and then merge. Afterwards we had to rename them again, to "Label2" and "Description2", and merge again. We've also added, a new column that is the concatenation between "Description1", "Relation" and "Description2", so we have something like: "Girl plays Violin".

```
#Joined2
classes_df = classes_df.rename(columns={'Label': 'Label1', 'Description':
                                        'Description1'})
joined2_df = pd.merge(relations_df, classes_df, on='Label1')
classes_df = classes_df.rename(columns={'Label1': 'Label2', 'Description1':
                                        'Description2'})
joined2_df = pd.merge(joined2_df, classes_df, on='Label2')
joined2_df['FinalRelation'] = joined2_df['Description1']+' '
                              +joined2_df['Relation']+' '
                              +joined2_df['Description2']
```

The third one, “joined3”, is the merge between “joined” and “joined2”, on “ImageId”. “Label” was dropped, because it was redundant. And “Description” was renamed to “Class”. This dataframe holds every information, about each “ImageId”.

```
#Joined3
joined3_df = pd.merge(joined_df, joined2_df, on='ImageId')
joined3_df.drop(columns=['Label'], inplace=True)
joined3_df.rename(columns={'Description': 'Class'}, inplace=True)
```

2.6. Load Joined Dataframes to BigQuery

To finish, we created and loaded the “joined” dataframes, into tables in our dataset. Now, we have every table, ready to use, to develop the AppEngine app.

```
joined_ref = dataset_id+".joined"
client.delete_table(joined_ref, not_found_ok=True)
joined_table = bq.Table(joined_ref)
joined_table.schema = (
    bq.SchemaField('ImageId',        'STRING'),
    bq.SchemaField('Label',          'STRING'),
    bq.SchemaField('Description',    'STRING')
)
client.create_table(joined_table)
joined_load = client.load_table_from_dataframe(joined_df, joined_table)
```

3. AppEngine

3.1. Requirements

On top of the libraries defined in the “requirements.txt” supplied, we added “pandas==2.2.1”. Since we’re going to use it in the “main.py”.

3.2. Relations

For the “relations” python function, we queried the BigQuery dataset, “bdcc24-project1.openimages.relations2”. And selected the column “Relation”, and counted the occurrence of each “Relation”, labeling it as “Image Count”. We saved the results, as a dictionary. That is passed to the HTML template, “relations.html”.

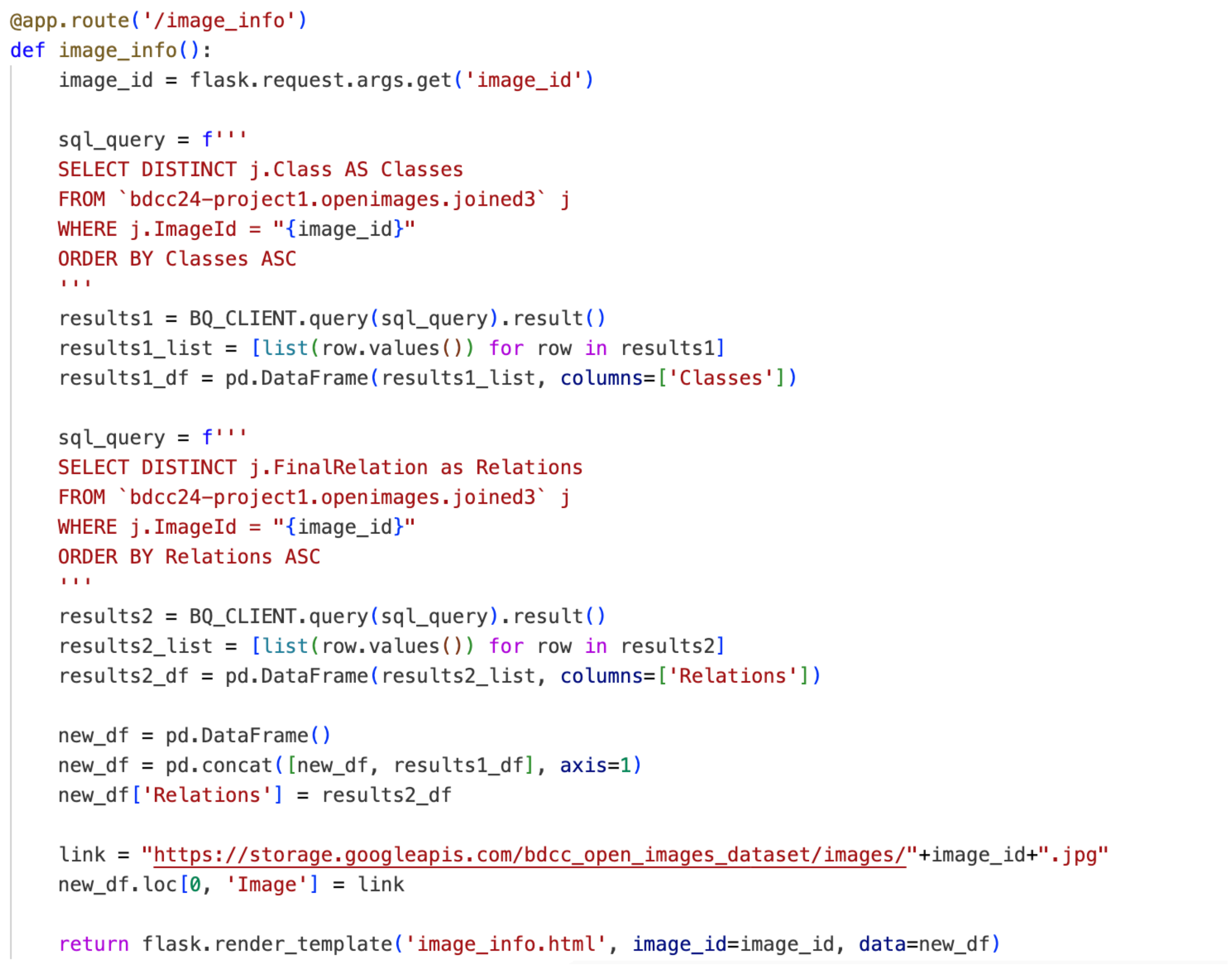
```python
@app.route('/relations')
def relations():
    results = BQ_CLIENT.query(
        '''
            SELECT Relation, COUNT(*) AS `Image count`
            FROM `bdcc24-project1.openimages.relations2`
            GROUP BY Relation
            ORDER BY Relation ASC
        '''
    ).result()
    relation_list = [{'Relation': row.Relation, 'Image count': row['Image count']} for row in results]

    return flask.render_template('relations.html', data={'relations': relation_list})
```

For the HTML script, “relations.html”, we created a table, with the columns “Relation” and “Image Count”. In the “Relation” column, we displayed every relation, where we redirected them to it’s “relation_search” page, with the default values, but the “relation”.

```html
<table>
    <tr>
        <th>Relation</th>
        <th>Image count</th>
    </tr>
    {% for relation in data.relations %}
    <tr>
        <td>
            <a href="/relation_search?relation={{ relation.Relation }}">{{ relation.Relation }}</a>
        </td>
        <td>
            {{ relation['Image count'] }}
        </td>
    </tr>
    {% endfor %}
</table>
```

3.3. Image Info

For the “image_info” python function, we queried the BigQuery dataset, “bdcc24-project1.openimages.joined3”, two times, generating two different dataframes. The first one, has the “Classes” and the second one, has the “Relations”, with the “ImageId”=image_id, selected by the user. We created a new dataframe, that is the merge between the first two dataframes. And we added a new column, “Image”, with only one observation, that is the link for the image file stored in the public storage bucket. This dataframe and the “image_id” are passed to the HTML template, “image_info.html”.



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

    sql_query = f'''
    SELECT DISTINCT j.Class AS Classes
    FROM `bdcc24-project1.openimages.joined3` j
    WHERE j.ImageId = "{image_id}"
    ORDER BY Classes ASC
    '''
    results1 = BQ_CLIENT.query(sql_query).result()
    results1_list = [list(row.values()) for row in results1]
    results1_df = pd.DataFrame(results1_list, columns=['Classes'])

    sql_query = f'''
    SELECT DISTINCT j.FinalRelation as Relations
    FROM `bdcc24-project1.openimages.joined3` j
    WHERE j.ImageId = "{image_id}"
    ORDER BY Relations ASC
    '''
    results2 = BQ_CLIENT.query(sql_query).result()
    results2_list = [list(row.values()) for row in results2]
    results2_df = pd.DataFrame(results2_list, columns=['Relations'])

    new_df = pd.DataFrame()
    new_df = pd.concat([new_df, results1_df], axis=1)
    new_df['Relations'] = results2_df

    link = "https://storage.googleapis.com/bdcc_open_images_dataset/images/"+image_id+".jpg"
    new_df.loc[0, 'Image'] = link

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

For the HTML script, “image_info.html”, we created a table, with only one row, with the columns “Classes”, “Relations” and “Image”. In the “Classes” column, we displayed every class, where we redirected them, to it’s “image_search”. In the “Relations” column, we displayed every relation, where we redirected them to it’s “relation_search” page. In the “Image” column, we displayed a clickable image, from the link.

```
<table>
    <tr>
        <th>Classes</th>
        <th>Relations</th>
        <th>Image</th>
    </tr>
    <tr>
        <td>
            {% for index, row in data.iterrows() %}
                {% if not row['Classes']|string == 'nan' %}
                    <a href="/image_search?description={{ row['Classes'] }}">
                        {{ row['Classes'] }}</a><br>
                {% endif %}
            {% endfor %}
        </td>


        <td>
            {% for index, row in data.iterrows() %}
                {% if not row['Relations']|string == 'nan' %}
                    {% set relation_split = row['Relations'].split(' ') %}
                    {% set class1 = relation_split[0] %}
                    {% set relation = relation_split[1] %}
                    {% set class2 = relation_split[2] %}
                    <a href="/relation_search?class1={{ class1 }}&relation={{ relation }}
                    &class2={{ class2 }}">{{ row['Relations'] }}</a><br>
                {% endif %}
            {% endfor %}
        </td>
        <td>
            <a href="{{ data.iloc[0]['Image'] }}" target="_blank">
                <img src="{{ data.iloc[0]['Image'] }}" style="width: 300px;">
            </a>
        </td>
    </tr>
</table>
```

3.4. Image Search

For the "image_search" python function, we queried the BigQuery dataset, "bdcc24-project1.openimages.joined". And selected the column "ImageId", with the "Description"=description, and only "image_limit" rows. Then, we converted the query to a dataframe, and added a column "Image", with the link, for every "ImageId". This dataframe and the "description", "image_limit" and "result_count" (length of dataframe) are passed to the HTML template, "image_search.html".

```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)

    sql_query = f'''
    Select ImageId from `bdcc24-project1.openimages.joined`
    WHERE Description = "{description}"
    ORDER BY ImageId
    LIMIT {image_limit}
    '''
    results = BQ_CLIENT.query(sql_query).result()

    results_list = [list(row.values()) for row in results]
    results_df = pd.DataFrame(results_list, columns=['ImageId'])
    results_df['Image'] = results_df['ImageId'].apply(lambda x: generate_link(x))
    results_count = len(results_df)

    return flask.render_template('image_search.html', description=description,
    image_limit=image_limit, results_count=results_count, data=results_df)
```

For the HTML script, “image_search.html”, we created a table, with the columns, “ImageId” and “Image”. In the “ImageId” column, we displayed every image identifier, where we redirected them to it’s “image_info”. In the “Image” column, we displayed a clickable image, from the link.

```html
<table>
    <tr>
        <th>ImageId</th>
        <th>Image</th>
    </tr>
    {% for index, row in data.iterrows() %}
    <tr>
        <td>
            <a href="/image_info?image_id={{ row['ImageId'] }}">{{ row['ImageId'] }}</a>
        </td>
        </td>
        <td>
            <a href="{{ row['Image'] }}" target="_blank">
                <img src="{{ row['Image'] }}" style="width: 300px;">
            </a>
        </td>
    </tr>
    {% endfor %}
</table>
```

3.5. Relation Search

For the “relation_search” python function, we queried the BigQuery dataset, “bdcc24-project1.openimages.joined3”. And selected the columns “ImageId”, “Description1”=class1, “Relation”=relation, “Description2”=class2 and only “image_limit” rows. Then, we converted the query to a dataframe, and added a column "Image", with the link, for every "ImageId". This dataframe and the "class1", "relation", "class2", "image_limit" and "result_count" (length of dataframe) are passed to the HTML template, "relation_search.html".

```
@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)

    sql_query = f'''
    SELECT DISTINCT ImageId, Description1, Relation, Description2
    FROM `bdcc24-project1.openimages.joined3`
    WHERE Description1 LIKE "{class1}"
    AND Relation LIKE "{relation}"
    AND Description2 LIKE "{class2}"
    ORDER BY ImageId
    LIMIT {image_limit}
    '''

    results = BQ_CLIENT.query(sql_query).result()

    results_list = [list(row.values()) for row in results]
    results_df = pd.DataFrame(results_list, columns=['ImageId', 'Class 1', 'Relation',
                                                     'Class 2'])
    results_df['Image'] = results_df['ImageId'].apply(lambda x: generate_link(x))
    results_count = len(results_df)

    return flask.render_template('relation_search.html', class1=class1, relation=relation,
    class2=class2, image_limit=image_limit, results_count=results_count, data=results_df)
```

For the HTML script, "relation_search.html", we created a table, with the columns "ImageId", "Class 1", "Relation", "Class 2" and "Image". In the "ImageId" column, we displayed every image identifier, where we redirected every image identifier to it's "image_info". In the "Class 1" and "Class 2" columns, we displayed every class, where we redirect them to it's "image_search". In the "Relations" column, we displayed every relation, where we redirected them to it's "relation_search" page. In the "Image" column, we displayed a clickable image, from the link.

```
<table>
    <tr>
        <th>ImageId</th>
        <th>Class 1</th>
        <th>Relation</th>
        <th>Class 2</th>
        <th>Image</th>
    </tr>
    {% for index, row in data.iterrows() %}
    <tr>
        <td>
            <a href="/image_info?image_id={{ row['ImageId'] }}">{{ row['ImageId'] }}</a>
        </td>
        <td>
            <a href="/image_search?description={{ row['Class 1'] }}">{{ row['Class 1'] }}</a>
        </td>
        <td>
            <a href="/relation_search?&relation={{ row['Relation'] }}">{{ row['Relation'] }}</a>
        </td>
        <td>
            <a href="/image_search?description={{ row['Class 2'] }}">{{ row['Class 2'] }}</a>
        </td>
        <td>
            <a href="{{ row['Image'] }}" target="_blank">
                <img src="{{ row['Image'] }}" style="width: 300px;">
            </a>
        </td>
    </tr>
    {% endfor %}
</table>
```

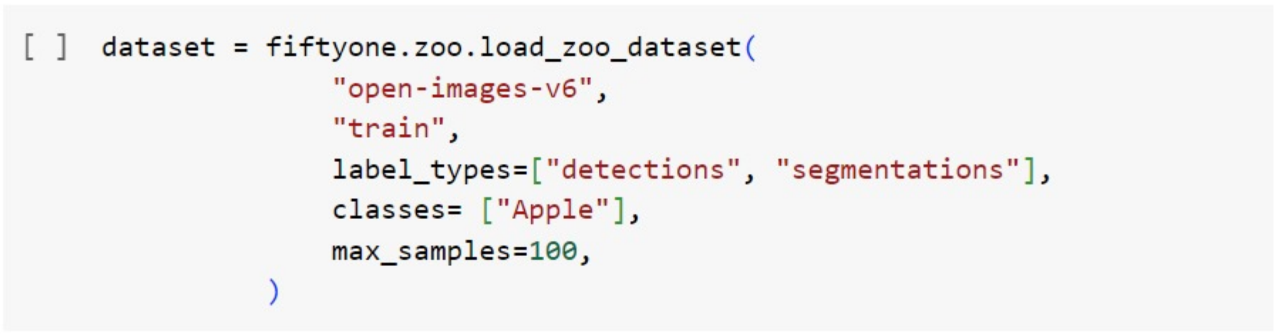
4. TF Lite Model

4.1. Data Preparation

The first step in creating our own TF Lite Model, consisted in the preparation of data. This entailed the creation of a distinct bucket that would be the focus for the storage of images in our project, in our case the bucket created was “bdcc24_open_images_dataset”. The images were found and downloaded through the BigQuery dataset using the open-source tool FiftyOne. The 10 image classes consisted of types of foods, more precisely: Apple, Orange, Hamburger, Peach, Pizza, Sandwich, Tart, Milk, Ice Cream and Pasta. The TFLite model creation process was mainly done using the Vertex AI feature and a Google Colab Notebook was used to implement the FiftyOne tool.

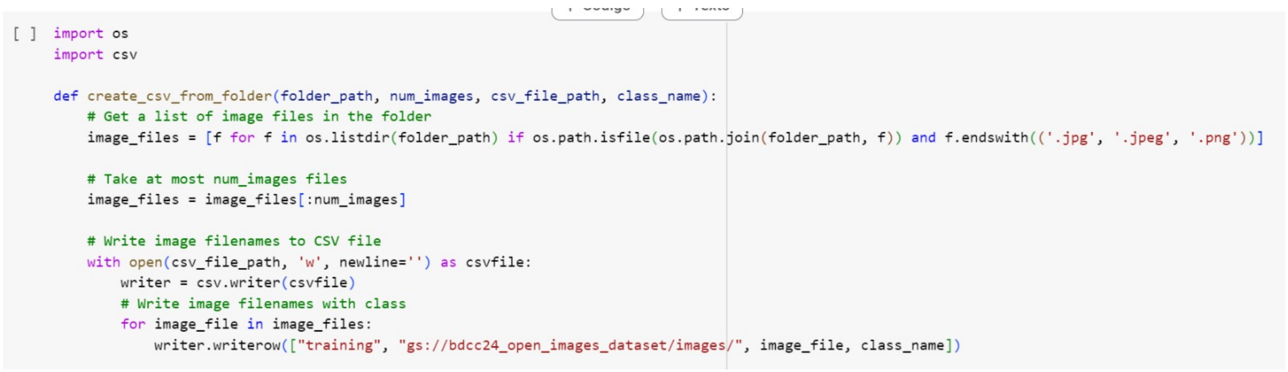
4.2. Google Colab

In the Google Colab notebook we first installed and imported FiftyOne using a pip install. 10 different datasets codes were done with a max sample of 100 images. This helped with getting a clearer and more precise process in the gathering of photos, since all the images would gather in the same data folder and the use of one dataset with 10 classes and a max sample of 1000 images would not divide the 100 images per class evenly.



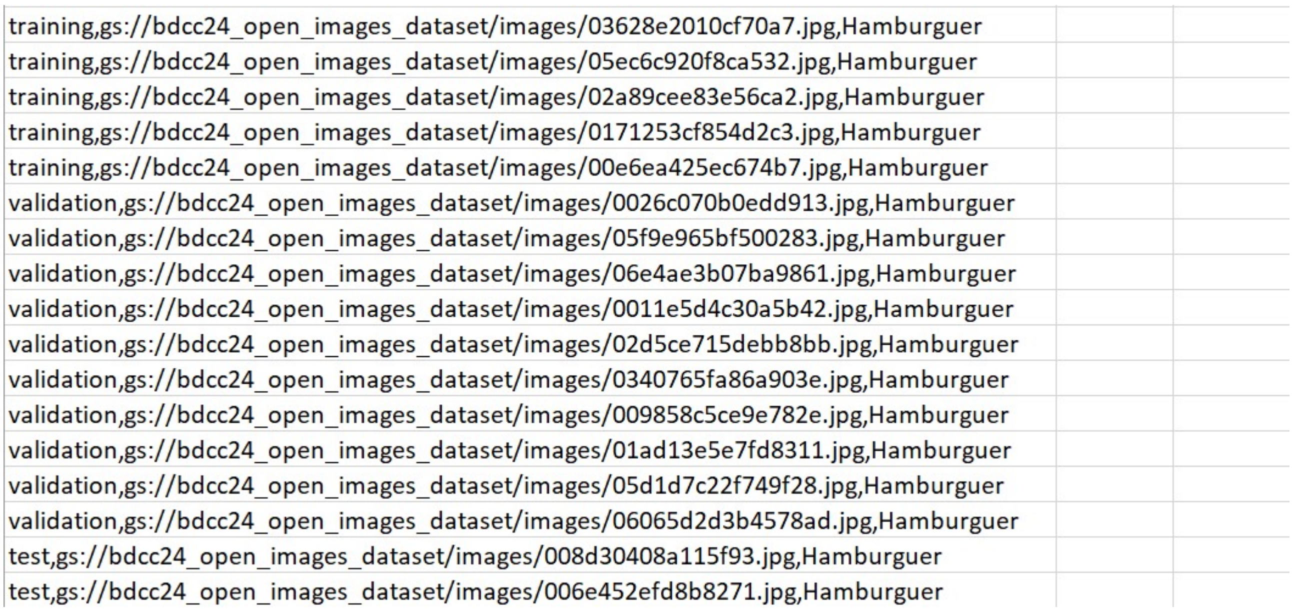
```
[ ]  dataset = fiftyone.zoo.load_zoo_dataset(
                  "open-images-v6",
                  "train",
                  label_types=["detections", "segmentations"],
                  classes= ["Apple"],
                  max_samples=100,
              )
```

After gathering all the photos in a data file, a CSV was made. It had the purpose of containing all available images in the format that was demonstrated by Prof. Eduardo Marques in the “static/tflite/dataset.csv” file. The code used created a CSV file that would display each image per cell with the same structure as the previously referenced file (ML_USED,GCS_FILE_PATH,[LABEL]).



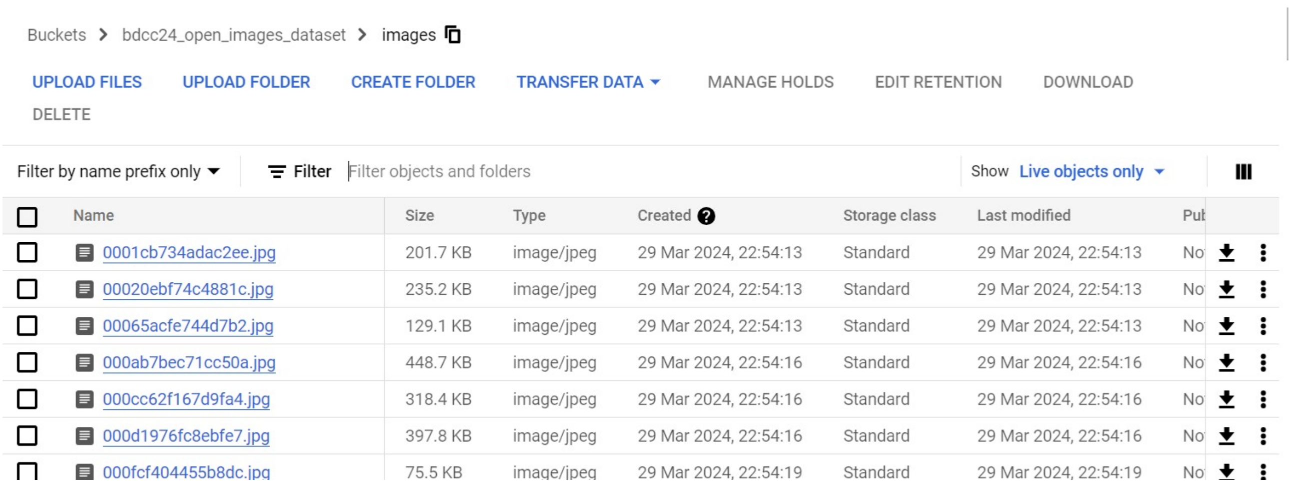
```
[ ]  import os
     import csv

     def create_csv_from_folder(folder_path, num_images, csv_file_path, class_name):
         # Get a list of image files in the folder
         image_files = [f for f in os.listdir(folder_path) if os.path.isfile(os.path.join(folder_path, f)) and f.endswith(('.jpg', '.jpeg', '.png'))]

         # Take at most num_images files
         image_files = image_files[:num_images]

         # Write image filenames to CSV file
         with open(csv_file_path, 'w', newline='') as csvfile:
             writer = csv.writer(csvfile)
             # Write image filenames with class
             for image_file in image_files:
                 writer.writerow(["training", "gs://bdcc24_open_images_dataset/images/", image_file, class_name])
```

After the CSV was made, it was inspected to make sure that it had a custom data split of 80:10:10 in terms of number of images to be considered in the training, validation or test set.



```
training,gs://bdcc24_open_images_dataset/images/03628e2010cf70a7.jpg,Hamburguer
training,gs://bdcc24_open_images_dataset/images/05ec6c920f8ca532.jpg,Hamburguer
training,gs://bdcc24_open_images_dataset/images/02a89cee83e56ca2.jpg,Hamburguer
training,gs://bdcc24_open_images_dataset/images/0171253cf854d2c3.jpg,Hamburguer
training,gs://bdcc24_open_images_dataset/images/00e6ea425ec674b7.jpg,Hamburguer
validation,gs://bdcc24_open_images_dataset/images/0026c070b0edd913.jpg,Hamburguer
validation,gs://bdcc24_open_images_dataset/images/05f9e965bf500283.jpg,Hamburguer
validation,gs://bdcc24_open_images_dataset/images/06e4ae3b07ba9861.jpg,Hamburguer
validation,gs://bdcc24_open_images_dataset/images/0011e5d4c30a5b42.jpg,Hamburguer
validation,gs://bdcc24_open_images_dataset/images/02d5ce715debb8bb.jpg,Hamburguer
validation,gs://bdcc24_open_images_dataset/images/0340765fa86a903e.jpg,Hamburguer
validation,gs://bdcc24_open_images_dataset/images/009858c5ce9e782e.jpg,Hamburguer
validation,gs://bdcc24_open_images_dataset/images/01ad13e5e7fd8311.jpg,Hamburguer
validation,gs://bdcc24_open_images_dataset/images/05d1d7c22f749f28.jpg,Hamburguer
validation,gs://bdcc24_open_images_dataset/images/06065d2d3b4578ad.jpg,Hamburguer
test,gs://bdcc24_open_images_dataset/images/008d30408a115f93.jpg,Hamburguer
test,gs://bdcc24_open_images_dataset/images/006e452efd8b8271.jpg,Hamburguer
```

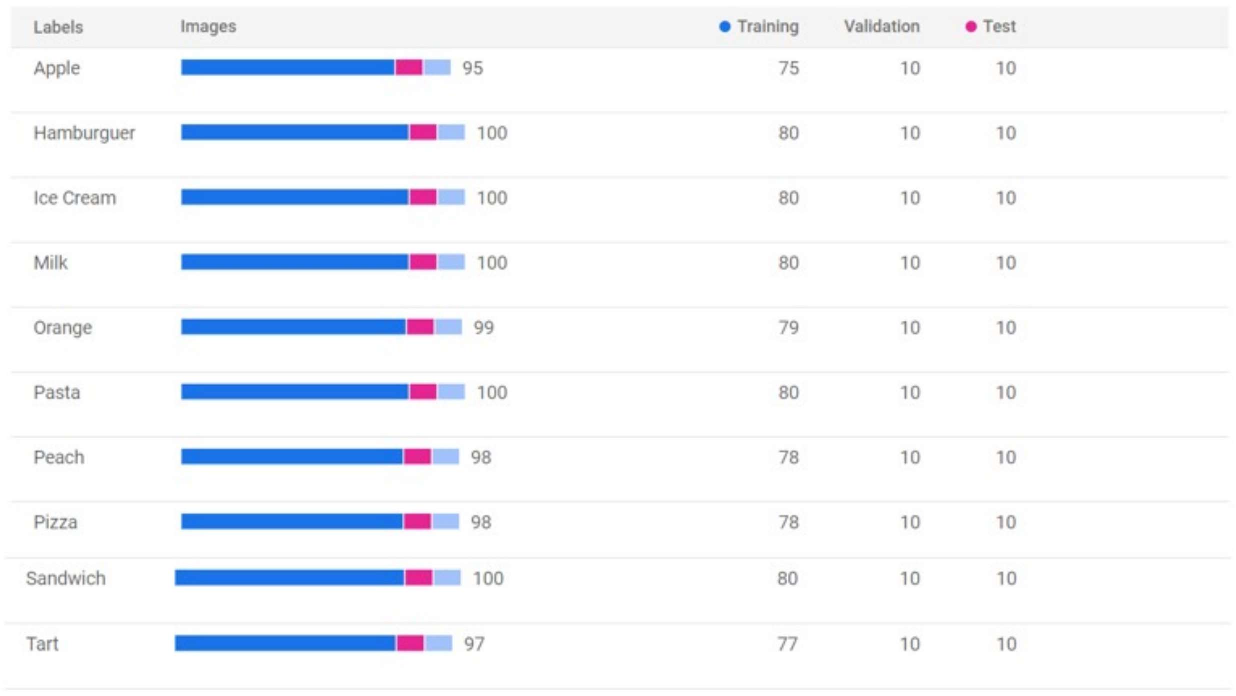
Finally, the data file containing all of the images was zipped and stored locally in an images files, which was then uploaded in the "bdcc24_open_images" bucket.



Name	Size	Type	Created	Storage class	Last modified	Pul
0001cb734adac2ee.jpg	201.7 KB	image/jpeg	29 Mar 2024, 22:54:13	Standard	29 Mar 2024, 22:54:13	No
00020ebf74c4881c.jpg	235.2 KB	image/jpeg	29 Mar 2024, 22:54:13	Standard	29 Mar 2024, 22:54:13	No
00065acfe744d7b2.jpg	129.1 KB	image/jpeg	29 Mar 2024, 22:54:13	Standard	29 Mar 2024, 22:54:13	No
000ab7bec71cc50a.jpg	448.7 KB	image/jpeg	29 Mar 2024, 22:54:16	Standard	29 Mar 2024, 22:54:16	No
000cc62f167d9fa4.jpg	318.4 KB	image/jpeg	29 Mar 2024, 22:54:16	Standard	29 Mar 2024, 22:54:16	No
000d1976fc8ebfe7.jpg	397.8 KB	image/jpeg	29 Mar 2024, 22:54:16	Standard	29 Mar 2024, 22:54:16	No
000fcf404455b8dc.jpg	75.5 KB	image/jpeg	29 Mar 2024, 22:54:19	Standard	29 Mar 2024, 22:54:19	No

4.3. Vertex AI

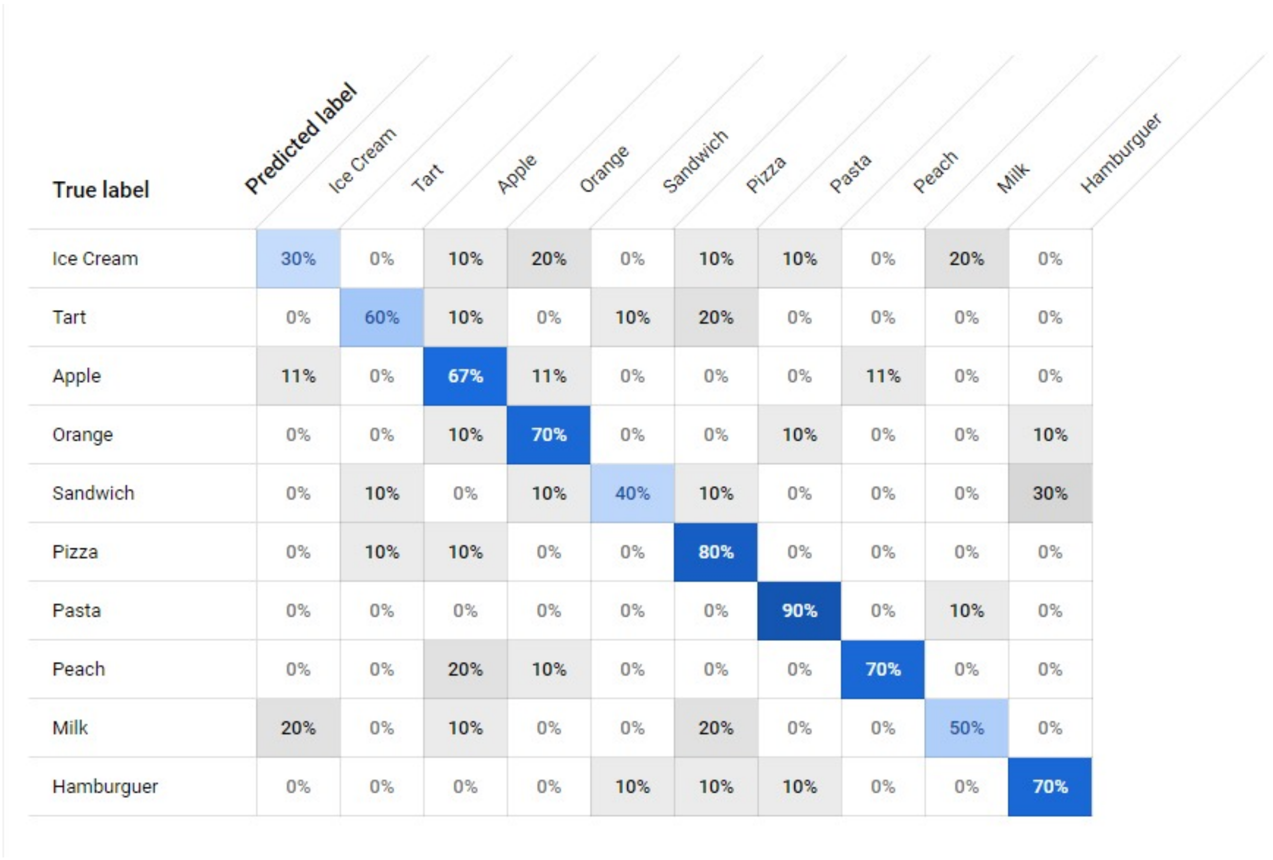
After creating the CSV file and storing the images in the bucket we followed the Vertex AI and TF Lite creation model tutorial. This consisted of creating a dataset on the Vertex AI window in the Google Cloud Console, with the purpose of image classification with a single label. After importing the CSV file from the computer and finding a cloud storage place for it, the dataset is created after it processes the images. Analysis of the Vertex AI shows us that a total of 14 images were unable to import data due to errors, mainly different classes containing the same picture.



Labels	Images	● Training	Validation	● Test
Apple	95	75	10	10
Hamburguer	100	80	10	10
Ice Cream	100	80	10	10
Milk	100	80	10	10
Orange	99	79	10	10
Pasta	100	80	10	10
Peach	98	78	10	10
Pizza	98	78	10	10
Sandwich	100	80	10	10
Tart	97	77	10	10

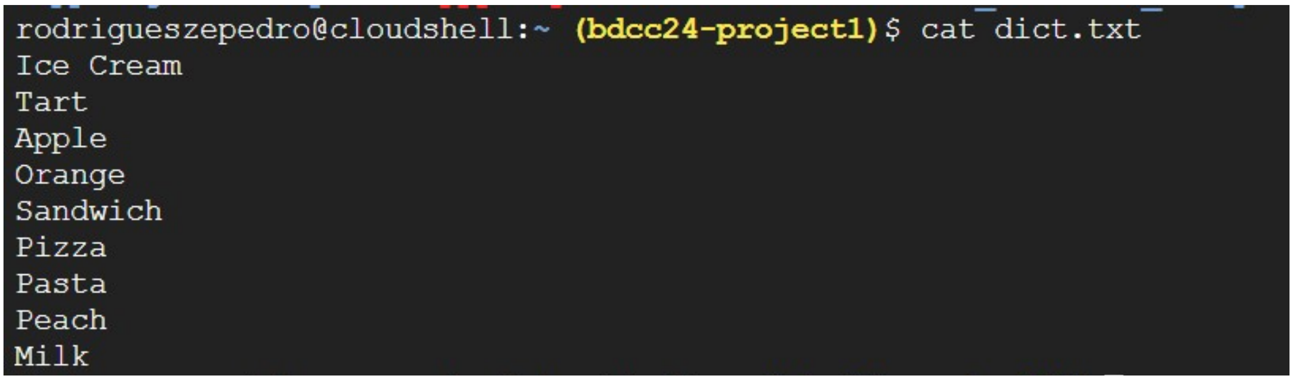
The training method used was AutoML which trains high quality models with minimal effort and machine learning expertise. The model was used in Edge which is used for on-prem and on-devise use but has lower accuracy. The training option goal is the one with best trade-off that has medium accuracy but has a lower package size. The budget set for training in compute-hours was 1, setting the estimated time of completion to be 1 hour. The early stopping was enabled in order to save the most amount of budget possible.

After training the model the evaluation showed that at a 0.5 confidence level the model had a precision of 82% and a recall of 47%. Looking at the confidence matrix, surprisingly the class with the lowest predictions correct was milk. Despite having all its images available this low percentage could be due to the images used in the test set having some of the other food types around it.



True label \ Predicted label	Ice Cream	Tart	Apple	Orange	Sandwich	Pizza	Pasta	Peach	Milk	Hamburguer
Ice Cream	30%	0%	10%	20%	0%	10%	10%	0%	20%	0%
Tart	0%	60%	10%	0%	10%	20%	0%	0%	0%	0%
Apple	11%	0%	67%	11%	0%	0%	0%	11%	0%	0%
Orange	0%	0%	10%	70%	0%	0%	10%	0%	0%	10%
Sandwich	0%	10%	0%	10%	40%	10%	0%	0%	0%	30%
Pizza	0%	10%	10%	0%	0%	80%	0%	0%	0%	0%
Pasta	0%	0%	0%	0%	0%	0%	90%	0%	10%	0%
Peach	0%	0%	20%	10%	0%	0%	0%	70%	0%	0%
Milk	20%	0%	10%	0%	0%	20%	0%	0%	50%	0%
Hamburguer	0%	0%	0%	0%	10%	10%	10%	0%	0%	70%

Finally, the model was exported and stored in a bucket. It was then unzipped in the cloud shell releasing two files a "dict.txt" and a "model.tflite". The cloud shell editor was then used to replace their respective files in the "main.py" that was given as a starting point by Prof. Eduardo Marques.



```
rodrigueszepedro@cloudshell:~ (bdcc24-project1)$ cat dict.txt
Ice Cream
Tart
Apple
Orange
Sandwich
Pizza
Pasta
Peach
Milk
```

5. Conclusion

In summary, this project has been dedicated to building an AppEngine application utilizing the Open Images dataset and Google Cloud's resources. By integrating BigQuery for data management and TensorFlow Lite models trained with Vertex AI for image classification, we've developed an application for users to learn from image data. Throughout this paper we explained every step towards the objectives defined by the Professor.