Project 1

Big Data & Cloud Computing - CC4093

Pedro Eduardo Nogueira da Mota - UP201805248

Departamento de Ciência de Computadores

Faculdade de Ciências da Universidade do Porto

April 2022

Contents

Preface

In this project, we programmed an AppEngine app that provides information about images taken from the Open Images dataset. The app also employs a TensorFlow model for image classification derived using AutoML Vision. The identifier of the Google Cloud project is **bdcc-project1-346914** and the app is available at https://bdcc-project1-346914.oa.r.appspot.com and https://docker-app-2tdegsengq-oa.a.run.app. This report is intended to describe the work that has been done.

1 Defining the BigQuery data set

For the initial development of the app, we used the *bdcc22project.openimages* BigQuery data set, but after implementing the missing endpoints, we defined our own BigQuery data set *bdcc-project1-346914.openimages* and used it in our app. For that purpose, we wrote a Python script (*app/create_dataset.py*), following a similar script done in the forth lab class. In this script, we create the three tables (*classes, image_labels* and *relations*), define their schema and load the data from the *csv* files into them. As an ilustration, we show in figure 1 the whole process for the *relations* table.

```
relations = pd.read_csv("./data/relations.csv")

table_name = PROJECT_ID + '.' + dataset_name + '.relations'
print('Creating table ' + table_name)

# Delete the table in case you're running this for the second time
client.delete_table(table_name, not_found_ok=True)

# Create the table
table = bq.Table(table_name)
table.schema = (
        bq.SchemaField('ImageID',  'STRING'),
        bq.SchemaField('Label1',   'STRING'),
        bq.SchemaField('Relation', 'STRING'),
        bq.SchemaField('Label2',   'STRING')
)
client.create_table(table)

# Load the data
print('Loading data into ' + table_name)
load_job = client.load_table_from_dataframe(relations, table)

while load_job.running():
  print('waiting for the load job to complete')
  time.sleep(1)

if load_job.errors == None:
  print('Load complete!')
else:
  print(load_job.errors)
```

Figure 1

2 Implementation of the missing endpoints

Regarding the missing endpoints - *relations*, *relations_search*, *image_info* and *image_search_multiple* - in order to implement them, we took inspiration from the already implemented endpoints. After the implementation of each missing endpoint, we made extensive tests to check if our app's behaviour was equal to the demo app's behaviour. No differences were found.
Sections 2.1, 2.2, 2.3 and 2.4 show how we have implemented the missing endpoints, in the order presented before.

2.1 The *relations* endpoint

The *relations* endpoint is supposed to show the available relations' types and the number of images (in the dataset) that contain that particular relation.

In order to implement the *relations* endpoint, we gain inspiration from the *classes* endpoint.

In the *main.py* python file, the function corresponding to the *relations* endpoint is equal to the *classes* endpoint, but with the query shown in figure 2.

```
results = BQ_CLIENT.query(
'''
    Select Relation, COUNT(*) AS NumImages
    FROM `bdcc-project1-346914.openimages.relations`
    GROUP BY Relation
    ORDER BY Relation
''').result()
data = dict(results=results)
```

Figure 2

The *relations*' HTML template is also similar to the *classes*' HTML template, except the title of the first column is *Relation*, instead of *Description*.

2.2 The *relations_search* endpoint

The *relations_search* endpoint is supposed to search for images by relation.

The query made and the data used in the HTML template is shown in figure 3.

As recommended by the teacher, we used the *LIKE* operator. In this way, we can use % to match any number of characters, as in the demo app, for instance.

Regarding the HTML template, we simply create a table with 5 columns - *ImageId, Class 1, Relation, Class2* and *Image* - and, then, we simply iterate through the data, and append the corresponding results to each column.

```python
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 r.ImageId, f.Description, r.Relation, s.Description
    FROM `bdcc-project1-346914.openimages.relations` r
    JOIN `bdcc-project1-346914.openimages.classes` f ON (f.Label = Label1)
    JOIN `bdcc-project1-346914.openimages.classes` s ON (s.Label = Label2)
    WHERE f.Description LIKE '{0}' AND r.Relation LIKE '{1}' AND s.Description LIKE '{2}'
    ORDER BY r.ImageId
    LIMIT {3}
'''.format(class1, relation, class2, image_limit)
).result()

data = dict(results=results,
            class1=class1,
            relation=relation,
            class2=class2,
            image_limit=image_limit)
```

Figure 3

2.3 The *image_info* endpoint

The *image_info* endpoint is supposed to give information about a single image.

The queries made and the data used in the HTML template is shown in figure 4.

```python
image_id = flask.request.args.get('image_id')

classes = BQ_CLIENT.query(
'''
    SELECT Description
    FROM `bdcc-project1-346914.openimages.image_labels`
    JOIN `bdcc-project1-346914.openimages.classes` USING(Label)
    WHERE ImageId = '{0}'
    ORDER BY Description
'''.format(image_id)
).result()

relations = BQ_CLIENT.query(
'''
    SELECT f.Description, Relation, s.Description
    FROM `bdcc-project1-346914.openimages.relations`
    JOIN `bdcc-project1-346914.openimages.classes` f ON(f.Label=Label1)
    JOIN `bdcc-project1-346914.openimages.classes` s ON(s.Label=Label2)
    WHERE ImageId = '{0}'
    ORDER BY f.Description, Relation, s.Description
'''.format(image_id)
).result()

data = dict(image_id=image_id,
            classes=classes,
            relations=relations)
```

Figure 4

With the first query, we get only the classes that are part of the image. Then, with the second query, we get the relations that are part of our image, in a similar way.

In the HTML template, we made a table with 3 columns - *Classes*, *Relations* and *Image*.

Then, for each column, we loop the corresponding field in the data and append the results, as in the demo app.

2.4 The *image_search_multiple* endpoint

The *image_search_multiple* endpoint is supposed to allow the user to search for images based on multiple labels.

The queries made and the data used in the HTML template is shown in figure 5.

```
descriptions = flask.request.args.get('descriptions').split(',')
image_limit = flask.request.args.get('image_limit', default=10, type=int)

results = BQ_CLIENT.query(
'''
    SELECT ImageID, ARRAY_AGG(Description) AS Descriptions
    FROM `bdcc-project1-346914.openimages.image_labels`
    JOIN `bdcc-project1-346914.openimages.classes` USING(Label)
    WHERE Description IN UNNEST({0})
    GROUP BY ImageID
    ORDER BY array_length(Descriptions) DESC, ImageID ASC
    LIMIT {1}
'''.format(descriptions, image_limit)
).result()

data = dict(descriptions=descriptions,
            image_limit=image_limit,
            results=results)
```

Figure 5

The query simply get the *ImageIDs* from the images that have, at least, one description in the labels given by the user. The query also aggregates all descriptions of each particular image.

In the corresponding HTML template, we made a table with 3 columns - *ImageID*, *Classes* and *Image*. Then, for each column, we loop the corresponding field in the data and append the results, as in the demo app.

3 Deriving a TensorFlow model with AutoML

In the begining, we played around with the demo version and the initial code as to get familiar with the problem. After this, we looked for the available descriptions and choose 10 different classes to our new model. The classes choosen were *Apple, Boat, Bicycle, Bird, Ball, Car, Dog, Cat, Fish* and *House*.

With the classes choosen, we then wrote a script (*app/generate_automl.py*) to get 100 random images from each class. The script basically initializes a BigQuery client and, for each class c, it executes the query shown in figure 6 in order to get 100 random different ImageIDs.

```
results = BQ_CLIENT.query(
'''
    Select ImageID
    FROM `bdcc-project1-346914.openimages.image_labels`
    JOIN `bdcc-project1-346914.openimages.classes` USING (Label)
    WHERE Description = '{0}'
    ORDER BY RAND()
    LIMIT 100
'''.format(c)
).result()
```

Figure 6

Then, for each class, we just append to the *automl.csv* file the first 80 images as part of the training set, the following 10 as the verification set and the last 10 as the test set, as in the initial given *automl.csv* file, as in the original *automl.csv* file. Luckily, the sets constructed are disjoint.

Before we start training our model, there's a few thing we still needed to do, such as enabling the AutoML and Cloud Storage APIs and also creating a Cloud Storage bucket to store the the sample images and the *automl.csv* file. In order to do this, we followed this tutorial.

To create the buckets, we simply did as in figure 7.

```
export PROJECT_ID=bdcc-project1-346914
gsutil mb -p ${PROJECT_ID} -c regional -l us-central1 gs://${PROJECT_ID}-vcm/
export BUCKET=${PROJECT_ID}-vcm
```

Figure 7

With the bucket created, we can copy the images into our bucket - figure 8.

```
echo automl.csv | grep -o "gs.*.jpg" | gsutil -m cp -I gs://${BUCKET}/img/
```

Figure 8

One problem that emerged was that our generated *automl.csv* had directionaries to the images in the public data set, so we needed to redirect to the images in our bucket. In order to do this, we executed the command shown in figure 9.

```
sed -i -- "s/gs:\/\/bdcc_open_images_dataset\/images\//gs:\/\/${BUCKET}\/img\//g" automl.csv
```

Figure 9

And then we upload the *automl.csv* file into the bucket.

With this all done, we could finally train our multi-label classification model. We choose to optimize the model for the best trade-off. The training process took about 1 hour. Regarding the model perfomance, figure 10 shows the precision and recall obtained and figure 11 shows the confusion matrix obtained, for the testing set.

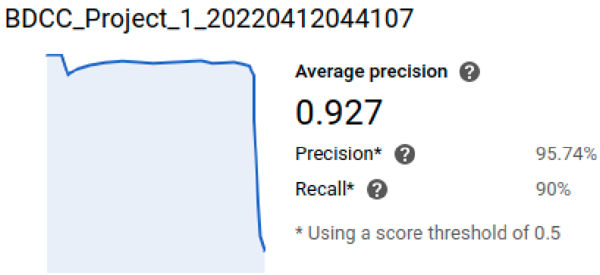

Figure 10

True Label \ Predicted Label	Cat	Ball	Dog	House	Car	Fish	Apple	Boat	Bicycle	Bird
Cat	100%	-	-	-	-	-	-	-	-	-
Ball	-	80%	-	-	-	-	10%	-	10%	-
Dog	-	-	90%	-	10%	-	-	-	-	-
House	-	-	-	100%	-	-	-	-	-	-
Car	-	-	-	-	100%	-	-	-	-	-
Fish	-	-	-	-	-	100%	-	-	-	-
Apple	-	-	-	-	-	-	100%	-	-	-
Boat	-	-	-	-	-	-	-	100%	-	-
Bicycle	-	-	-	-	10%	-	-	-	90%	-
Bird	-	-	10%	-	-	10%	-	-	-	80%

Figure 11

Finally, through the web interface, we exported our model as a TensorLite package and used it in our app, by substituting the files in *app/static/tflite* to the new ones.

And, it's done!

To illustrate the differences between the classifications from the model deployed in the demo app and the model one in our app, figures 12, 13 show the obtained classifications for an image of a cat, respectively.

1 images classified with a minimum confidence level of **0.05**.

<table>
<tr><th>Filename</th><th>Classifications</th><th>Image</th></tr>
<tr><td>Cat.jpg</td><td>
<table>
<tr><th>Class</th><th>Confidence</th></tr>
<tr><td>Lion</td><td>0.22</td></tr>
<tr><td>Butterfly</td><td>0.21</td></tr>
<tr><td>Duck</td><td>0.13</td></tr>
<tr><td>Frog</td><td>0.09</td></tr>
<tr><td>Elephant</td><td>0.08</td></tr>
<tr><td>Tortoise</td><td>0.08</td></tr>
<tr><td>Chicken</td><td>0.06</td></tr>
<tr><td>Bull</td><td>0.06</td></tr>
<tr><td>Monkey</td><td>0.05</td></tr>
</table>
</td><td></td></tr>
</table>

Figure 12

1 images classified with a minimum confidence level of **0.05**.

<table>
<tr><th>Filename</th><th>Classifications</th><th>Image</th></tr>
<tr><td>Cat.jpg</td><td>
<table>
<tr><th>Class</th><th>Confidence</th></tr>
<tr><td>Cat</td><td>0.82</td></tr>
<tr><td>Dog</td><td>0.22</td></tr>
<tr><td>Car</td><td>0.12</td></tr>
<tr><td>Boat</td><td>0.10</td></tr>
<tr><td>House</td><td>0.09</td></tr>
<tr><td>Bird</td><td>0.08</td></tr>
<tr><td>Bicycle</td><td>0.08</td></tr>
<tr><td>Apple</td><td>0.07</td></tr>
<tr><td>Fish</td><td>0.06</td></tr>
<tr><td>Ball</td><td>0.06</td></tr>
</table>
</td><td></td></tr>
</table>

Figure 13

4 Additional challenges

4.1 Using the Cloud Vision API

The objective is to develop an alternative app endpoint for image classification that makes use of label detection through the Google Cloud Vision API using the corresponding Python client API.

In order to do this, we started by setting up a service account, through the web interface, with *Owner*'s permissions. Then, we created a key and downloaded the key as a *json* file. This key will allow us to identify to the Google Cloud service and we use it in the initialization of the Google Cloud Vision client, as shown in figure 14.

```
logging.info('Initialising Vision Client')
from google.cloud import vision
from google.oauth2 import service_account
credentials = service_account.Credentials. \
    from_service_account_file('./static/key/bdcc-project1-346914-a1fd72420bb8.json')
client = vision.ImageAnnotatorClient(credentials=credentials)
```

Figure 14

We can finally create the new endpoint, which we shown in figure 15.

```
@app.route('/label_detection', methods=['POST'])
def label_detection():
    files = flask.request.files.getlist('files')
    results = []
    if len(files) > 1 or files[0].filename != '':
        for file in files:
            blob = storage.Blob(file.filename, APP_BUCKET)
            blob.upload_from_file(file, blob, content_type=file.mimetype)
            blob.make_public()

            response = client.annotate_image({
                'image':
                    {'source':
                        {'image_uri': 'https://storage.googleapis.com/{0}/{1}'.format(APP_BUCKET.name, file.filename)}},
                'features':
                    [{'type_': vision.Feature.Type.LABEL_DETECTION}]
            })

            response = AnnotateImageResponse.to_json(response)

            response = json.loads(response)

            annotations = []
            for annotation in response['labelAnnotations']:
                annotations.append(dict(label=annotation['description'], score='%.2f'%annotation['score']))

            results.append(dict(bucket=APP_BUCKET,
                                filename=file.filename,
                                annotations=annotations))

    data = dict(bucket_name=APP_BUCKET.name, results=results)
    return flask.render_template('label_detection.html', data=data)
```

Figure 15

For each file, we simply copy it to the bucket, in order to use the Cloud Vision API. Then, we send a resquest to annotate the image, with the path to the image and, also, stating that the task is to do label detection. After we get the response, we parse it as a *json* object and then we loop through the annotations and simply gather the *description* and *score* fields.

The structure of the HTML template is identical to the *image_classify* endpoint. We also had to add in the *requeriments.txt* file the Google Cloud Vision requeriment - *google-cloud-vision*, in order to do the deployment.

4.2 Defining a Docker image for the app

The objective of this task is too define our own container for the app, using a *Dockerfile*. The written dockerfile is shown in figure 16.

```
1  FROM python:3.10-slim
2
3  COPY . /docker-app
4
5  WORKDIR /docker-app
6
7  RUN pip3 install -r requirements.txt
8
9  ENV FLASK_APP=main.py
10 ENV FLASK_RUN_HOST=0.0.0.0
11 ENV FLASK_RUN_PORT=8080
12
13 ENV GOOGLE_CLOUD_PROJECT=bdcc-project1-346914
14
15 CMD [ "python3", "-m" , "flask", "run"]
```

Figure 16

In line 1, we define our starting base image, which is a light-version of Python, version 3.10. In line 3, we copy the source code to the a new directory */docker-app*. In line 5, we set */docker-app* as the working directory for the following instructions. Then, in line 7, we install all the dependencies, which are all reported in the requirements.txt file. Lines 9 to 11, we define environment variables, for the flask application, respectively, the entry python file, the host and the port for our application. In line 13, we define another environment variable, which will be used in our application, which is the project ID. Finally, in line 15, we define the command to run within the container, in order to start the app.

To run the app through the Cloud Shell, we executed the commands shown in figure 17.

```
docker build --tag docker-app .
docker run -d -p 8080:8080 docker-app
```

Figure 17

The first command builds the Docker image from the Dockerfile (in the current directory) and the second one runs the built image.

To run the app through the Cloud Run, we follow this tutorial, namely the *Building with a Dockerfile* section, where it shows how to build a Docker image using just a Dockerfile, without requiring a separate build config file. Figure 18 shows the two executed commands.

```
gcloud builds submit --tag gcr.io/bdcc-project1-346914/docker-app
gcloud run deploy --image gcr.io/bdcc-project1-346914/docker-app --platform managed
```

Figure 18

Similarly as in the Cloud Shell, the first command builds the Docker image using the Dockerfile (in the current directory). The second command runs the Docker image that we built before. The URL for the deployed app: `https://docker-app-2tdegsengq-oa.a.run.app/`.