FACULTY OF SCIENCES OF THE
UNIVERSITY OF PORTO

MASTER'S DEGREE IN INFORMATION SECURITY

Big Data & Cloud Computing

1st Practical Assignment

David Capela Rui Fernandes

May, 2022

Abstract

This report describes the first practical assignment of the Big Data & Cloud Computing course of the Master's Degree in Information Security at the Faculty of Sciences of the University of Porto.

This assignment aims at developing an AppEngine app that not only provides information about images taken from the Open Images dataset, but also employs a TensorFlow model for image classification derived using AutoML Vision.

In this report, we briefly describe the application we developed and discuss the decisions we made.

Contents

List of Figures

Listings

1 Introduction

This project consists in developing an AppEngine app that provides information about images from the Open Images dataset. In addition, the app makes use of a TensorFlow model for image classification derived from AutoML Vision. As such, the work is divided into the following parts:

1. Defining a BigQuery dataset from CSV files.
2. Implementing the app endpoints to query data stored in BigQuery, and reference the corresponding image files stored in a public storage bucket.
3. Deriving our own TensorFlow model with AutoML, which replaces the one provided with the application.
4. Developing an endpoint for image classification that uses label detection through the Google Cloud Vision API.
5. Defining our own Docker container for the app.

Project ID	`big-data-project1-347618`
AppEngine URL	`https://big-data-project1-347618.uc.r.appspot.com`
Docker container URL on Cloud Run	`https://demo-app-uqhmrwookq-oa.a.run.app`

Structure of the Report

The remainder of this work is structured as follows:

– In Section 2, **Data Model & Application Endpoints**, we describe the data model, the available endpoints and how they were implemented.

– In Section 3, **TensorFlow Model Using AutoML**, we explain how we our own TensorFlow model with AutoML, replacing the one provided with the application.

– In Section 4, **Additional Challenges**, the additional challenges are addressed, which include image classification through the Google Cloud Vision API, and defining a Docker container for the app.

– Finally, Section 5, **Conclusions**, concludes the report.

2 Data Model & Application Endpoints

The BigQuery dataset contains the following three tables:

– `classes`: Contains the label and a brief description of each

– `image_labels`: Contains the image ID and the respective class label.

– `relations`: Contains the image ID, and the relationship between the respective classes.

This data model is illustrated in Figure 1.

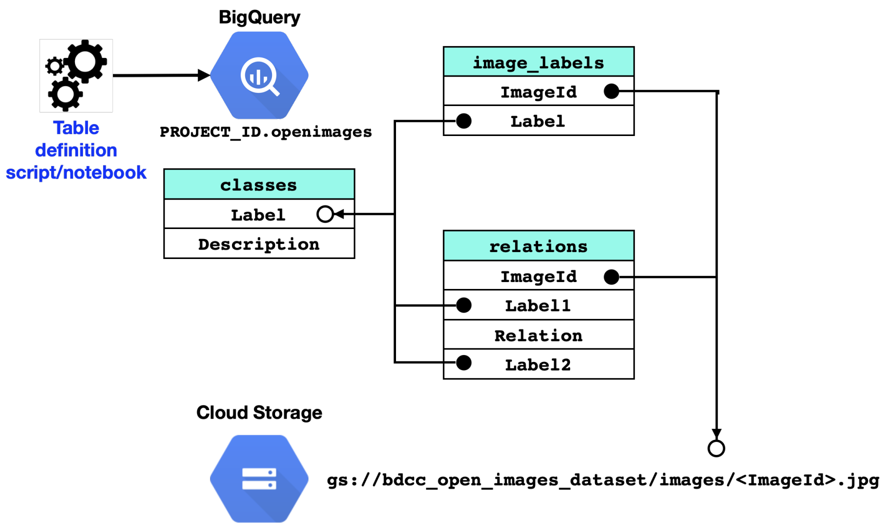

Figure 1: Data model

For initial development of the app, we used the `bdcc22project.openimages` BigQuery dataset. However, we then defined our own BigQuery data set. To do so, we wrote a Python script, as shown in

To access information contained in the dataset, a few endpoints were implemented. Each endpoint performs one or more BigQuery queries, and the returned data is used to create an HTML page that displays such data to the user.

The following application endpoints were already implemented:

– `classes`: List image labels.

– `image_search`: Search for images based on a single label.

– `image_classify_classes`: List available classes for image classification.

– `image_classify`: Use a TensorFlow model to classify images.

This being said, we only had to implement the following endpoints:

– `image_info`: Get information for a single image.

– `relations`: List relation types.

– `relations_search`: Search for images by relation.

– `image_search_multiple`: Search for images based on multiple labels.

The implementation of these endpoints is described in detail in the following sections, and snippets of code are included.

2.1 Image Info Endpoint

This endpoint lists information about a single image, *i.e.* a list of all relations and classes, given its ID. To do so, we wrote two queries: the first one selects the classes by joining the `image_labels` and `classes` tables using the label and filtering by those that have the requested ID. By doing so, we end up with the description of all the classes of the image in question.

In addition, the second query selects existing relationships between between any two classes by joining the `relations` and `classes` tables and then filtering by those that have the requested ID.

Having said this, the implementation of this endpoint is shown in Listing 2.

2.2 Relations Endpoint

This endpoint returns the relations between images. To do so, the query selects all relations, counts the number of images that satisfy each relation, groups them, and, finally, sorts them alphabetically. Thus, we end up with the number of images that satisfy each relation.

The implementation of this endpoint is shown in Listing 3.

2.3 Relation Search Endpoint

This endpoint search for images that satisfy a given relation (*e.g. Man holds Guitar*, where *Man* and *Guitar* are the classes and *holds* is the relation between them). In this case, the classes can be specified or default (denoted with the symbol `%`), and the limit of images to be displayed can also be specified.

To do so, the query selects `ImageId`, `Class1`, `Relation` and `Class2` by joining the `relations` the `classes` tables when the relations and the classes are the ones specified.

It is worth noting that this query makes use of the `LIKE` SQL operator such that, if no class parameters are supplied, the query uses the default `%`, returning image that satisfies the relationship, regardless of the classes involved in it.

The implementation of this endpoint is shown in Listing 4.

2.4 Image Search Multiple Endpoint

This query selects all images that correspond to at least one of the classes specified by the user, as well as the number of classes that matched. This search is performed in the `image_labels` table, as well as the classes table, using the `Label` attribute. To do so, we use two different functions:

- `ARRAY_AGG` is used to return an array with the classes associated with the said image.
- `UNNEST` is used to to transform the previously mentioned array in order to check whether a description was contained in the values of that same table.

The implementation of this endpoint is shown in Listing 5.

3 TensorFlow Model Using AutoML

We started by creating an AutoML dataset using a Python script `copy_images.py` that copied relevant bucket images from the provided example bucket to ours. After having the images in our bucket, we had to generate the `automl.csv` file that stated the images we wanted the AutoML model to be trained with and for that we used the `create_automl.py` script. Finally, we uploaded the result file from the last script into the AutoML project in Google Cloud and waited for it to be finished so we could download the `tflite.model` file. This process is illustrated in Figure 2.

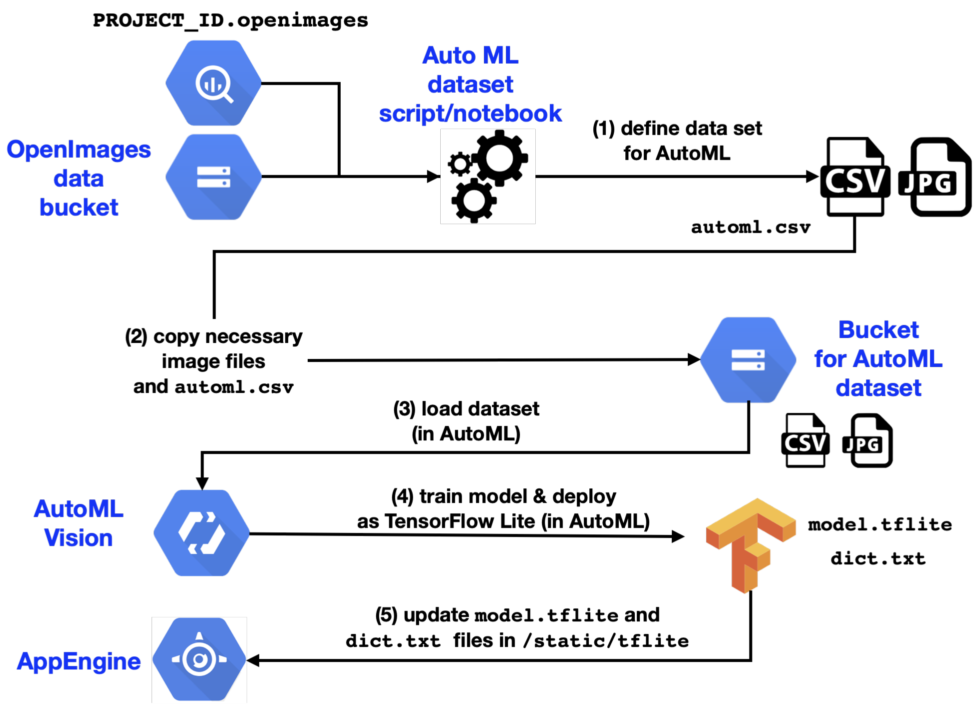
PROJECT_ID.openimages
OpenImages data bucket
Auto ML dataset script/notebook
(1) define data set for AutoML
CSV
JPG
automl.csv
(2) copy necessary image files and automl.csv
Bucket for AutoML dataset
(3) load dataset (in AutoML)
AutoML Vision
(4) train model & deploy as TensorFlow Lite (in AutoML)
model.tflite
dict.txt
(5) update model.tflite and dict.txt files in /static/tflite
AppEngine

Figure 2: TensorFlow model to classify images

4 Additional Challenges

In this section, we address the additional challenges, namely the implementation of an alternative endpoint for image classification using label detection through the Google Cloud Vision API, and the necessary steps we took in order to define our own Docker container for the application.

4.1 Label Detection Using Cloud Vision API

Besides creating our own TensorFlow model to perform image classification, we also implemented an endpoint that detects image labels by using Google's Cloud Vision API. The implementation of this endpoint can be seen in Listing 6.

4.2 Docker Image

Before defining the Docker container, we have to change `host='127.0.0.1'` to `host='0.0.0.0'` in the main function, that is, the `main` function is as follows:

```python
if __name__ == '__main__':
    # When invoked as a program.
    logging.info('Starting app')
    app.run(host='0.0.0.0', port=8080, debug=True)
```

This happens because the web server running in the container is listening for connections on port `8080` of the loopback network interface (`127.0.0.1`). Thus, it will only respond to HTTP requests originating from the container itself. To accept connections from outside of the container, we have to bind it to the `0.0.0.0` IP address.

Another thing to keep in mind is that we have to specify the `GOOGLE_CLOUD_PROJECT` environment variable, otherwise the project ID cannot be determined. This being said, the Dockerfile is as follows:

```dockerfile
# Python image to use.
FROM python:3.8

# Set the working directory to /app
WORKDIR /app

# Copy the requirements file used for dependencies
COPY requirements.txt .
```

```
# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Copy the rest of the working directory contents into the container at /app
COPY . .

# Set the GOOGLE_CLOUD_PROJECT environment variable
ENV GOOGLE_CLOUD_PROJECT=big-data-project1-347618

# Define network ports for the container to listen on at runtime
EXPOSE 8080

# Run main.py when the container launches
ENTRYPOINT ["python", "main.py"]
```

Listing 1: Dockerfile

We can run the app using Docker with the following commands:

```
$ docker build -t bdcc22 .
$ docker run -d -p 8080:8080 bdcc22
```

Finally, the application was deployed using Google Cloud Run, an is now available at `https://demo-app-uqhmrwookq-oa.a.run.app`. However, it is worth noting that sometimes we get a ”Service Unavailable” error when we try to access the web page. After trying different region parameters (namely, `europe-west1`, `europe-west6` and `us-central1`), the error remained, and due to time constraints, we were not able to look further into this issue. Regardless, given the fact that we can run the application with the command `docker run -d -p 8080:8080 bdcc22`, we know that the Dockerfile is well defined.

5 Conclusions

This project has been extremely valuable in terms of understanding the theoretical concepts taught in class, its challenges, and how they can be applied to real-world applications using big data and cloud computing.

As a matter of fact, we were able to implement all of the endpoints and create our own TensorFlow model with AutoML Vision, which allows us to perform image classification.

A Image Info Endpoint

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

    results_classes = BQ_CLIENT.query(
        '''
        SELECT Description
        FROM `big-data-project1-347618.dataset1.labels`
        JOIN `big-data-project1-347618.dataset1.classes` USING(label)
        WHERE ImageId = '{0}'
        ORDER BY Description ASC
    '''.format(image_id)
    ).result()

    results_relations = BQ_CLIENT.query(
        '''
        SELECT C1.Description as Class1, R.Relation, C2.Description as Class2
        FROM `big-data-project1-347618.dataset1.relations` R
        JOIN `big-data-project1-347618.dataset1.classes` C1 ON (R.label1=C1.label)
        JOIN `big-data-project1-347618.dataset1.classes` C2 ON (R.label2=C2.label)
        WHERE R.ImageId = '{0}'
    '''.format(image_id)
    ).result()

    data = dict(description=image_id,
                classes=results_classes,
                relations=results_relations
                )
    logging.info('image_info: image_id={}, classes={}, relations={}'.format(
        image_id, results_classes.total_rows, results_relations.total_rows))
    return flask.render_template('image_id.html', data=data)
```

Listing 2: Implementation of the `image_info` endpoint

B Relations Endpoint

```python
@app.route('/relations')
def relations():
    results = BQ_CLIENT.query(
        '''
        SELECT Relation, COUNT(*) As NumImages
        FROM `big-data-project1-347618.dataset1.relations`
        GROUP BY Relation
        ORDER BY Relation ASC
    ''').result()

    logging.info('classes: results={}'.format(results.total_rows))

    data = dict(results=results)

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

Listing 3: Implementation of the `relations` endpoint

C Relation 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 R.ImageId, C1.Description as Class1, R.Relation, C2.Description as Class2
        FROM `big-data-project1-347618.dataset1.relations` R
        JOIN `big-data-project1-347618.dataset1.classes` C1 ON (R.label1=C1.label)
        JOIN `big-data-project1-347618.dataset1.classes` C2 ON (R.label2=C2.label)
        WHERE R.Relation LIKE '{0}'
        AND C1.Description LIKE '{1}'
        AND C2.Description LIKE '{2}'
        ORDER BY R.ImageId
        LIMIT {3}
    '''.format(relation, class1, class2, image_limit)
    ).result()

    logging.info('relation_search: limit={}, results={}'
                 .format(image_limit, results.total_rows))
    data = dict(class1=class1,
                class2=class2,
                relation=relation,
                image_limit=image_limit,
                results=results)
    return flask.render_template('relation_search.html', data=data)
```

Listing 4: Implementation of the relation_search endpoint

D Image Search Multiple Endpoint

```python
@app.route('/image_search_multiple')
def image_search_multiple():
    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), COUNT(Description)
        FROM big-data-project1-347618.dataset1.labels
        JOIN big-data-project1-347618.dataset1.classes USING(label)
        WHERE Description IN UNNEST({0})
        GROUP BY ImageId
        ORDER BY Count(Description) DESC, ImageId
        LIMIT {1}
    '''.format(descriptions, image_limit)
    ).result()

    logging.info(
        'image_search_multiple: descriptions={} image_limit={} results={}'.format(
            descriptions, image_limit, results.total_rows)
    )
    data = dict(descriptions=descriptions,
                image_limit=image_limit,
                results=results)
    return flask.render_template(
        'image_search_multiple.html',
        data=data,
        descriptions=descriptions,
        image_limit=image_limit,
        total=results.total_rows,
        description_len=len(descriptions)
    )
```

Listing 5: Implementation of the image_search_multiple endpoint

E Label Detection Using Cloud Vision API

```python
@app.route('/cloud_vision', methods=['POST'])
def cloud_vision():
    files = flask.request.files.getlist('files')
    min_confidence = 1

    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)

            client = vision.ImageAnnotatorClient()
            image = vision.Image()
            image.source.image_uri = 'https://storage.googleapis.com/' + \
                APP_BUCKET.name + '/' + file.filename

            response = client.label_detection(image=image)

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

            classifications = response.label_annotations
            scores = list(map(lambda x: x.score, classifications))

            if min(scores) < min_confidence:
                min_confidence = min(scores)

            logging.info('cloud_vision: filename={} blob={} classifications={}, scores={}'
                         .format(file.filename, blob.name, classifications, scores))
            results.append(dict(bucket=APP_BUCKET,
                                filename=file.filename,
                                classifications=classifications))
```

```
data = dict(bucket_name=APP_BUCKET.name,
            min_confidence='{:.5f}'.format(min_confidence),
            results=results)
return flask.render_template('cloud_vision.html', data=data)
```

Listing 6: Implementation of the `cloud_vision` endpoint