BDCC_Report

April 10, 2021

1 Big Data and Cloud Computing - Report

Authors: David Maia (up201908521) and Miguel Tavares (up200902937)

GCP Project ID: bdcc-project1-309010

URL: https://bdcc-project1-309010.ew.r.appspot.com/

URL (deployed using docker): https://image-xhfxku6nlq-uc.a.run.app

2 Endpoints

image_search_multiple

This endpoint has an objective of receiving mulitple descriptions of images and a limit to the number of images presented. This query would select the ImageID (identifier of each image) and an array of its descriptions that belongs to the multiple descriptions requested. Ordering by the image with the most corresponding descriptions to the image with the least corresponding descriptions would give us the intended result.

```
[ ]: results = BQ_CLIENT.query(
         '''
         SELECT ImageID, ARRAY_AGG(DISTINCT Description) AS classes, COUNT(DISTINCT␣
       ↪Description) as c
         FROM `bdcc21project.openimages.classes`
         JOIN `bdcc21project.openimages.image_labels` USING(Label)
         WHERE Description IN UNNEST({0})
         GROUP BY ImageID
         ORDER BY c desc
         LIMIT {1}
         '''.format(descriptions, image_limit)
         ).result()
```

relations

This endpoint allows to show all the possible relations between images. The query selects all relations and counts how many images have it, groups them and orders for alphabetic order (Relation name).

```
[ ]: results = BQ_CLIENT.query(
         '''
             SELECT Relation, COUNT(*) AS NumImages
             FROM `bdcc21project.openimages.relations`
             GROUP BY Relation
             ORDER BY Relation asc
         ''').result()
```

relations_search

Search for images by relation (e.g. Girl plays Violin). This endpoint uses the operator LIKE so that if any of the three required parameters is not specified, it uses the default % and gives back the result as any. The query selects ImageId, Class1, Relation and Class2 by joining the table relations and the table classes twice (one for each class) when the relations and the classes are the requested ones.

```
[ ]: results = BQ_CLIENT.query(
         '''
         SELECT r.ImageId, c1.Description as Class1, r.Relation, c2.Description as␣
      ↪Class2
         FROM `bdcc21project.openimages.relations` r
         JOIN `bdcc21project.openimages.classes` c1 ON (r.Label1=c1.Label)
         JOIN `bdcc21project.openimages.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()
```

image_info

Get information for a single image. Providing only the imageID in this endpoint should be presented a list of all relations and classes. To simplify two queries were used: one for the classes and another for the relations. The first query selects all the Descriptions/Classes by joining two tables by the label and filtering by those that have the requested ImageId. The second query is based on the relations_search being the only diference that we only filter by the ImageId requested and not the classes.

```
[ ]: results_classes = BQ_CLIENT.query(
         '''
             SELECT Description
             FROM `bdcc21project.openimages.image_labels`
             JOIN `bdcc21project.openimages.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 `bdcc21project.openimages.relations` r
    JOIN `bdcc21project.openimages.classes` c1 ON (r.Label1=c1.Label)
    JOIN `bdcc21project.openimages.classes` c2 ON (r.Label2=c2.Label)
    WHERE r.ImageId = '{0}'
    '''.format(image_id)
    ).result()
```

3 TensorFlow dataset preparation notes

For this part it was requested to choose 10 classes. Below we can see the list of the chosen 10 classes.

```
[ ]: CLASSES =[
                ('Aircraft',),
                ('Bicycle',),
                ('Boat',),
                ('Bus',),
                ('Car',),
                ('Train',),
                ('Helicopter',),
                ('Motorcycle',),
                ('Truck',),
                ('Skateboard',)
     ]
```

This classes need to be in a dataframe so that we can use it after to select only images that belong to this classes.

```
[ ]: class_labels = spark.createDataFrame(data=CLASSES,schema=['Description'])
     class_labels.cache()
     class_labels.createOrReplaceTempView('class_labels')
     class_labels.printSchema()
     class_labels.show()
```

First, we use pyspark to join the dataframes image_labels, classes and class_labels. Now this dataframe has all the images that belong to one of our 10 chosen classes.

> To avoid a problem in AutoML which causes the dataset for each class to be much lower than 100 due to duplicates (e.g an image with a car and a boat can have both classes) is recommended to use *.dropDuplicates(["ImageId"])*. This way we ensure that each ImageID is our dataframe is unique.

```
[ ]: from pyspark.sql import functions as F
     from pyspark.sql.window import Window
     from pyspark.sql.functions import rank, col
     import pandas as pd
```

[ ]:
```
getimages = \
  image_labels.join(classes,'Label')\
  .join(class_labels,'Description')\
  .select('ImageId','Description')\
  .dropDuplicates(["ImageId"])\
  .orderBy('Description')
```

To select only 100 images for each class we use a Window partioned by Description and ordered by ImageId, this will cause the images to be ranked and separated by class. .filter(col('rank') <= 100) -> this filters the 100 first ImageId for each class In the end we converted this to pandas dataframe for the simplicity of usage.

[ ]:
```
window = \
  Window  \
  .partitionBy(getimages['Description'])\
  .orderBy(getimages['ImageId'])
```

[ ]:
```
final = \
  getimages.select('*', rank().over(window).alias('rank')) \
  .filter(col('rank') <= 100)\
  .toPandas()
```

The column rank was only for the previous operation, can now be removed from the dataframe

[ ]:
```
final = final.drop(columns='rank')
```

We used this last part so that the output file could be exactly as request but this is optional for AutoML since the program already separates the dataset with the default distribution of 80% to train, 10% to validate and 10% to test.

1. Add a new column "Train/Validation/Test"
2. For each class divide the dataset and add the corresponding fase to the column according to the distribution of 80% to train, 10% to validate and 10% to test.
3. Put the columns order as requested

[ ]:
```
final['train/test']=''
```

[ ]:
```
classes_list =[
          'Aircraft',
          'Bicycle',
          'Boat',
          'Bus',
          'Car',
          'Helicopter',
          'Motorcycle',
          'Skateboard',
          'Train',
          'Truck'
]
```

```
[ ]: for i in range(len(classes_list)):
         final.loc[i*100:i*100+79,'train/
      ↪test'][final['Description']==classes_list[i]] = 'TRAIN'
         final.loc[i*100+80:i*100+89,'train/
      ↪test'][final['Description']==classes_list[i]] = 'VALIDATION'
         final.loc[i*100+90:i*100+99,'train/
      ↪test'][final['Description']==classes_list[i]] = 'TEST'
```

```
[ ]: cols = final.columns.tolist()
     cols = cols[-1:] + cols[:-1]
     final = final[cols]
```

3.1 Move the data

Define the bucket that will be used to storage the necessary images to build the TF model

```
[ ]: MY_AUTOML_BUCKET='gs://bddc_train_transport'
```

Copy all the necessary images to the bucket using a for cycle and the gsutil command.

```
[ ]: for j in range(final.shape[0]):
       id = final.loc[[j],['ImageId']].values
       idf = id[0][0]
       !gsutil cp {BUCKET_URI}/images/{idf}.jpg {MY_AUTOML_BUCKET}/img/
       print(j)
```

The file will need the full path to container where the images are and not only the ImageID. To solve that issue we created a for loop to add the path to each ImageID.

```
[ ]: finalML = final
     j=0
     for j in range(finalML.shape[0]):
       id = finalML.loc[[j],['ImageId']].values
       idf = id[0][0]
       finalML.loc[[j],['ImageId']] = MY_AUTOML_BUCKET+'/img/'+idf+'.jpg'
```

The pandas dataframe now needs to be exported to the final automl.csv file that we want to use, the command .to_csv allows to export in a simple way that doesn't export header or index so that the file is exactly as requested.

```
[ ]: finalML.to_csv(MY_AUTOML_BUCKET+'/csv/automl.csv', header = False, index =␣
      ↪False)
```

3.2 Create TensorFlow model

In https://console.cloud.google.com/vision/dashboard?project=bdcc-project1-309010, the dataset was created using the automl.csv. Then the model was created and exported as TensorFlow to the web application.

3.3 Create Docker Image

Since AppEngine is enabled by containers internally, one way to deploy our application is using Docker. The first step is to create the Docker file which will define our environment.

This environment was imported from google container registry [1] and then ran with a virtual environment of python 3 (the default is python 2). The final step was to define the packages required to run our application in *requirement.txt* file.

The final instruction was to define the command to run a python application in a linux environment.

```
FROM gcr.io/google-appengine/python
RUN virtualenv /env -p python3
ENV VIRTUAL_ENV /env
ENV PATH /env/bin:$PATH
ENV GOOGLE_CLOUD_PROJECT=bdcc-project1-309010
ADD requirements.txt /app/requirements.txt
RUN pip install -r /app/requirements.txt

ADD . /app
WORKDIR /app

CMD [ "python3" , "main.py", "--host=0.0.0.0"]
```

[1] The gcr.io/google-appengine/python is a docker base image. This image can be used as the base image for running applications on Google App Engine Flexible, Google Kubernetes Engine, or any other Docker host.

This image is based on Ubuntu Xenial and contains packages required to build most of the popular Python libraries.

Once the Docker file defined, we need to build and submit it to the cloud server (the image shall be available in container registry):

```
[ ]: gcloud builds submit --tag gcr.io/PROJECT_ID/IMAGE_NAME
```

The next step is to deploy it to cloud run. Since our application needs more than 256 MB of memory (default value), it was deployed with 1 GB of memory. It is available in: https://image-xhfxku6nlq-uc.a.run.app