Big Data and Cloud Computing: Assignment 1 [*]

Emanuel Tomé [†], Vânia Guimarães [‡]

DCC-FCUP, 11th April 2021

1 Introduction

The project consists of creating a web application hosted in a cloud computing platform called Google App Engine. The application provides information of images that are stored in the bucket gs://bdcc_open_images_dataset using BigQuery. The application also have models to classify images, which was build using TensorFlow and AutoML Vision. Additionally, it is available an endpoint for classifying images which makes use of the Google Cloud Vision API and the corresponding Python client API. The application is deployed not only in the Google AppEngine, but also in a container explicitly defined for the application. In that case, the image-build process is defined by the build steps indicated in a Dockerfile and then the image is uploaded to Google Cloud Container Registry. The application is then run using the Google Cloud Run. The identifier of our Google Cloud Project is identified below as well as the urls for the AppEngine application and Docker image application:

- **Identifier of the Google Cloud Project**: tome-guima-project
- **URL for the AppEngine app**: https://tome-guima-project.uc.r.appspot.com
- **URL for the Docker image app**: https://tome-guima-run-nuxq7zecaa-uc.a.run.app

In this report the main steps for building the application are described. In Figure 1 is presented the diagram of the developed application.

2 BigQuery queries

In this section is presented an explanation of how the developed application works in terms of BigQuery queries. For a better following of the text, the relational representation of the database is presented in Figure 2.

2.1 Relations

We obtain all the relations available from table "relations" and with aggregated function count(*) and the option GROUP BY Relation we get the number of images in each relation. Then the results are kept in object "data" in dictionary format, to be read in html.

```python
@app.route('/relations')
def relations():
    results = BQ_CLIENT.query(
    '''
        Select Relation, COUNT(*) AS NumImages
        FROM `bdcc21project.openimages.relations`
        GROUP BY Relation
        ORDER BY Relation
    ''').result()
    logging.info('relations: results={}'.format(results.total_rows))
    data = dict(results=results)
    return flask.render_template('relations.html', data=data)
```

[*]This work was submitted on the framework of the course Big Data and Cloud Computing of the Master in Data Science.

[†]Emanuel Tomé is a student at the Faculty of Sciences of the University of Porto. Currently, he is enrolled in the 1st year of the Master's in Data Science (e-mail: up200702634@edu.fc.up.pt).

[‡]Vânia Guimarães is a student at the Faculty of Sciences of the University of Porto. Currently, she is enrolled in the 1st year of the Master's in Data Science (e-mail: up200505287@edu.fc.up.pt).

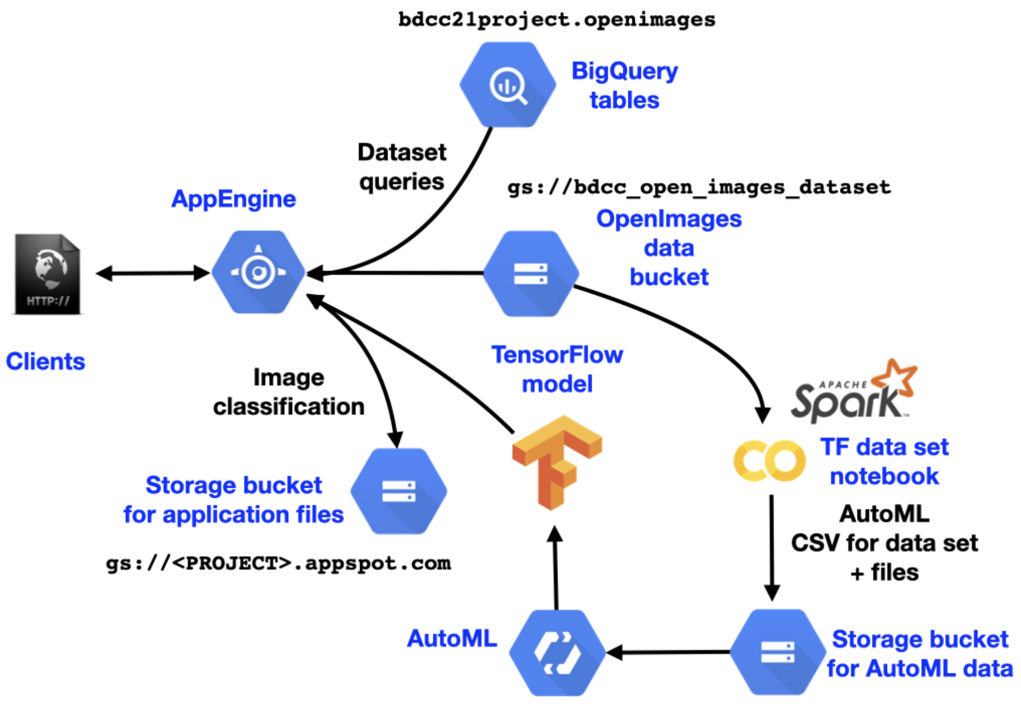


Figure 1: Diagram of the application.

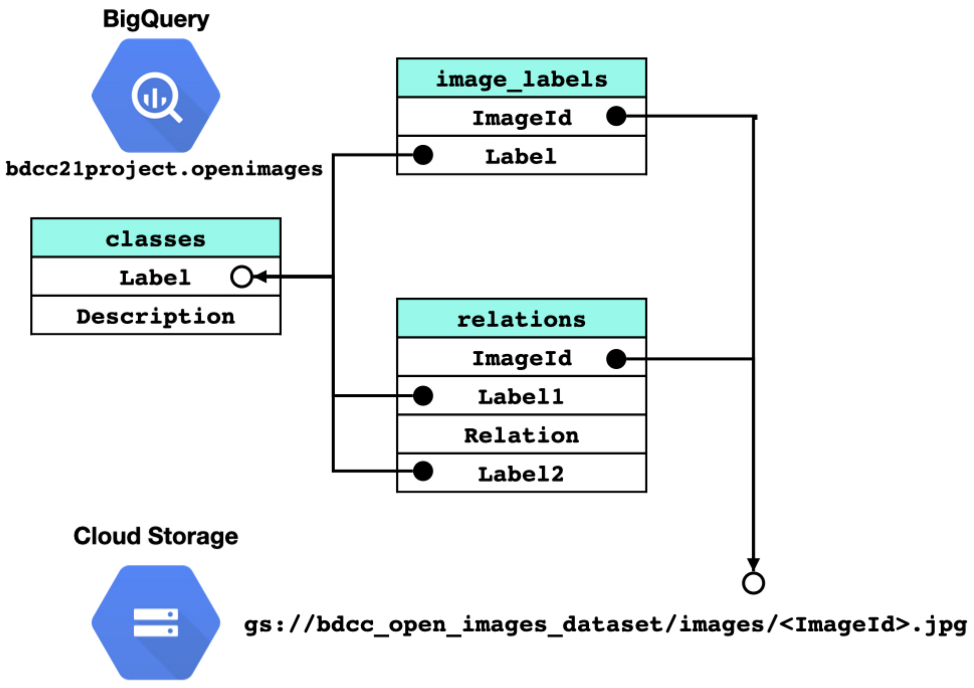


Figure 2: Data model.

2.2 Image_info

In the first query we obtain the distinct classes available on attribute "Description" from table "classes", related to the ImageId indicated by the client, which we find on table "image_labels".

On the second query we need the description associated to Label1, the relation and the description associated to Label2 related by the ImageId given by the user. Therefore it is necessary join classes and relations tables by the attribute Label. However we can not say that two attributes of a table are correlated to the same attribute of another. The alternative is give different names (l1 and l2) to classes table functioning, that way, as different tables. All the information is according the ImageId that is in table 'relations'. To get these outcomes in together in same row, we apply the operator CONCAT.

The results of two queries are in the dictionary named 'data'. In the key 'classes' are the results of the first query and in the key 'relations' are the results of the second.

```python
@app.route('/image_info')
def image_info():
    image_id = flask.request.args.get('image_id')
    classes = BQ_CLIENT.query(
    '''
        SELECT DISTINCT(c.Description) AS Classes
        FROM `bdcc21project.openimages.classes` c
        JOIN `bdcc21project.openimages.image_labels` il USING (Label)
        WHERE il.ImageId = '{0}'
        ORDER BY c.Description
    '''.format(image_id)).result()
    relations = BQ_CLIENT.query(
    '''
         SELECT DISTINCT(CONCAT(l1.Description,' ', r.Relation,' ', l2.Description)) AS
    Relations
        FROM `bdcc21project.openimages.relations` r
        LEFT JOIN `bdcc21project.openimages.classes` l1 ON (r.Label1 = l1.Label)
        LEFT JOIN `bdcc21project.openimages.classes` l2 ON (r.Label2 = l2.Label)
        WHERE r.ImageId = '{0}' and r.Relation IS NOT NULL
        ORDER BY Relations
    '''.format(image_id)).result()
    logging.info('image_info: image_id={}, classes={}, relations={}'.format(image_id, classes.
    total_rows, relations.total_rows))

    data = dict(image_id=image_id,
                classes=classes,
                relations=relations)
    return flask.render_template('image_info.html', data=data)
```

2.3 Relation_Search

The user have the possibility to choose three parameters, class1, relation and class2. According to same of this information, the application have to return the ImageId and the three parameters, chosen or not by the client. As we need the description of Label1 and Label2 we join the table 'classes' and table 'relations' by the attribute Label. Here we had the same reasoning as in the second query of endpoint/image_info. Moreover, as the client doesn't need to specify all the three categories we used the LIKE operator because accept the percent sign (%) which represents zero, one, or multiple characters.

```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, l1.Description, r.Relation, l2.Description
        FROM `bdcc21project.openimages.relations` r
        LEFT JOIN `bdcc21project.openimages.classes` l1 ON (r.Label1 = l1.Label)
        LEFT JOIN `bdcc21project.openimages.classes` l2 ON (r.Label2 = l2.Label)
        WHERE l1.Description LIKE '{0}' AND r.Relation LIKE '{1}' AND l2.Description LIKE
    '{2}'
        ORDER BY r.ImageId
        LIMIT {3}
    '''.format(class1, relation, class2, image_limit)).result()
    logging.info('relation_search: class1={}, relation={}, class2={}, limit={}, results={}'\
           .format(class1, relation, class2, image_limit, results.total_rows))
    data = dict(class1=class1,
                relation=relation,
                class2=class2,
                image_limit=image_limit,
                results=results)
    return flask.render_template('relation_search.html', data=data)
```

2.4 Image_search_multiple

In this endpoint the application should return the image with all, some or one classes chosen by the user. An image can have several classes, so we want to display the aggregated classes for each image. For that we use the operator ARRAY_AGG. As we are interested in images classified by all possible sets of classes determined by the user, the operator FROM UNNEST allows us to do it. Another information to display is the number of classes that image has, for that we use the aggregated function COUNT(). We are also interested in return the number of classes that the user choose, for that we use the operator ARRAY_LENGTH() to the array of their classes.

```
@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 il.ImageId, ARRAY_AGG(c.Description), count(*) as Total, ARRAY_LENGTH({0}) as size
        FROM `bdcc21project.openimages.classes` c
        JOIN `bdcc21project.openimages.image_labels` il USING(LABEL)
        WHERE c.Description IN (SELECT * FROM UNNEST ({0}))
        GROUP BY il.ImageId
        ORDER BY Total DESC, il.ImageId
        LIMIT {1}
    '''.format(descriptions, image_limit)
    ).result()
    logging.info('image_search_multiple: descriptions={} 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)
```

3 TensorFlow model using AutoML

In this section we describe the data set preparation to build a Convolution Neural Network for image classification using TensorFlow and AutoML. In Figure 1 is presented the flowchart of the followed procedure to build the TensorFlow model. We choose the following 10 classes: `Bread`, `Coin`, `Camera`, `Guitar`, `Laptop`, `Bus`, `Mushroom`, `Candle`, `Pizza` and `Sushi`.

3.1 Define the dataset to use using spark

In order to define the dataset, we started by construct a table class_labels with the chosen classes:

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

We then made the following query using spark in order to obtain 100 images of each class:

```
classes_selected = spark.sql('''
   WITH X
      AS
      (
          SELECT
            il.ImageID, c.Description, ROW_NUMBER() OVER(PARTITION BY c.Description ORDER BY il.ImageID) AS Num
          FROM
            classes c, image_labels il, class_labels cl
          WHERE
              c.Label=il.Label AND cl.Description=c.Description
          ORDER BY c.Description, il.ImageID
      )
      SELECT
```

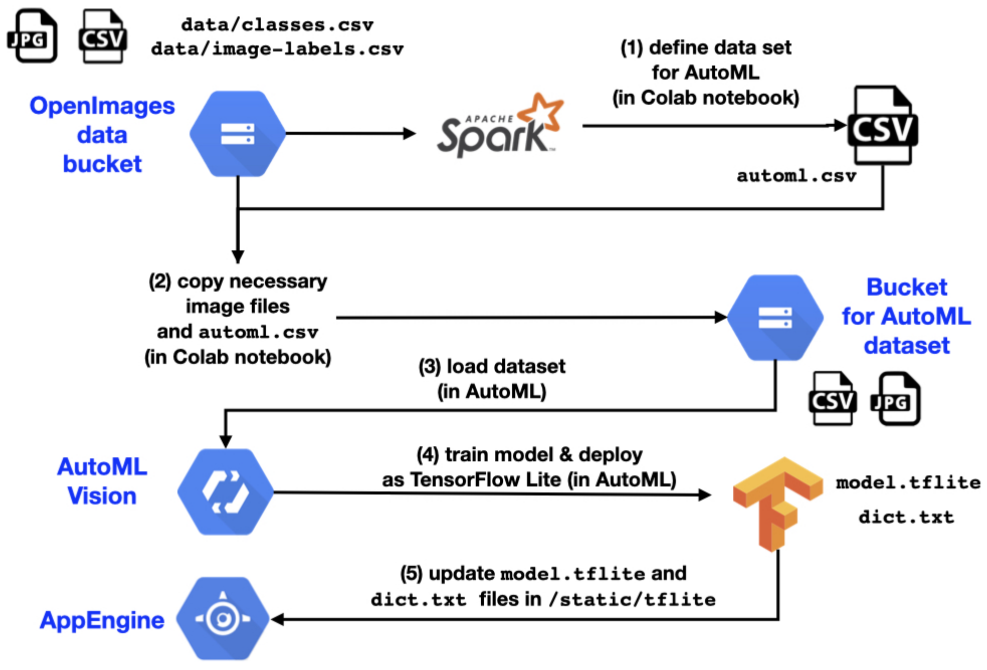

data/classes.csv
data/image-labels.csv
OpenImages data bucket
(1) define data set for AutoML (in Colab notebook)
automl.csv
(2) copy necessary image files and automl.csv (in Colab notebook)
Bucket for AutoML dataset
(3) load dataset (in AutoML)
AutoML Vision
(4) train model & deploy as TensorFlow Lite (in AutoML)
model.tflite
dict.txt
(5) update model.tflite and dict.txt files in /static/tflite
AppEngine

Figure 3: Flowchart to construct the TensorFlow model.

```
14          *
15        FROM
16          X
17        WHERE
18          Num <= 100
19 ''')
```

We defined a new table, X - lines 5 to 11, which has 3 attributes: `ImageID`, `Description` and `Num`, where `Num` is the line number for the combination of `ImageID` and `Description`. This table is constructed by joining tables classes and image_label by the attribute Label and the Description of tables classes and class_labels. Therefore, from this table we only kept the first 100 rows for each class (lines 13 to 18)

3.2 Put the data in a convenient bucket

After the data is selected, we then transferred the necessary images to train our TensorFlow model to our bucket. We then get the ImageId using Spark:

```
ImageId = classes_selected.select("ImageId").rdd.flatMap(lambda x: x).collect()
```

and them simply constructed a Python list with the full path of the images to be transferred:

```
filelist = ['gs://bdcc_open_images_dataset/images/' + Image + '.jpg' for Image in ImageId]
```

We then save the list of images to transfer to a local text file and then transferred then to our bucket using gsutil (line 5) with the parallel multi-threaded/multi-processing copy activated (flag -m) which took about 1 minute to transfer 1000 images for our bucket.

```
1 with open('filelist.txt', 'w') as f:
2     for item in filelist:
3         f.write("%s\n" % item)
4 
5 !cat filelist.txt | gsutil -m cp -I gs://tome-guima/images
```

We then constructed the csv file needed to import the images to train the TensorFlow model in the AutoML Vision of Google Cloud. For that propose, we build a list with the format required by AutoML Vision and then send it to our bucket using gsutil:

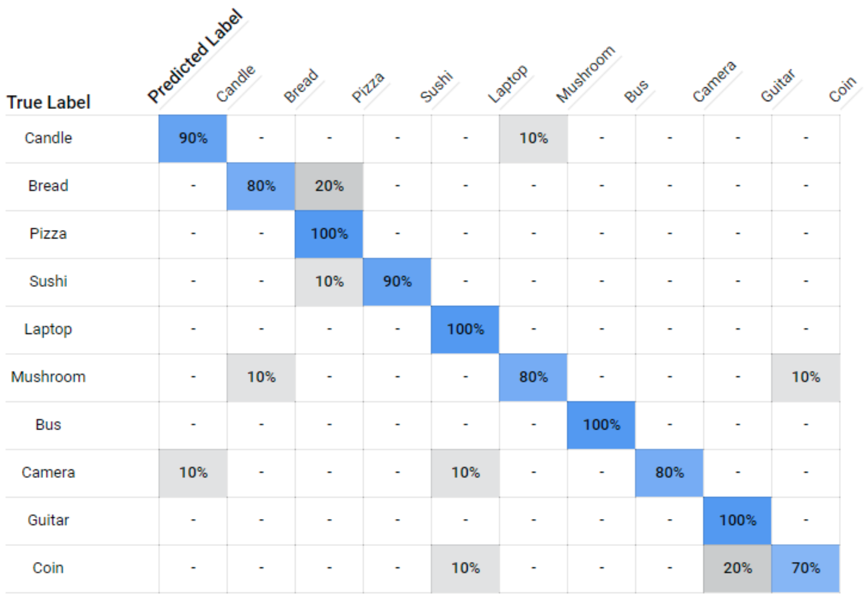
```
Description = classes_selected.select("Description").rdd.flatMap(lambda x: x).collect()
list_train=['TRAIN']*80+['VALIDATION']*10+['TEST']*10
filelist_2 = [list_train[k%100]+',gs://tome-guima/images/' + ImageId[k] + '.jpg,' +
    Description[k] for k in range(len(ImageId))]
```

```

With open('automl_selected.csv', 'w') as f:
    for item in filelist_2:
        f.write("%s\n" % item)

!gsutil cp automl_selected.csv gs://tome-guima/
```

We then build a single-label classification, training it during 3 hours. We obtained an Precision of 97.4% and a Recall of 75% for the text dataset. In Figure ?? we present the confusion matrix obtained for the test dataset. As one can state, there are four classes with a true positives rate of 100% and the class `Coin` is the one with the lower true positive rate (70%).



True Label \ Predicted Label	Candle	Bread	Pizza	Sushi	Laptop	Mushroom	Bus	Camera	Guitar	Coin
Candle	90%	-	-	-	-	10%	-	-	-	-
Bread	-	80%	20%	-	-	-	-	-	-	-
Pizza	-	-	100%	-	-	-	-	-	-	-
Sushi	-	-	10%	90%	-	-	-	-	-	-
Laptop	-	-	-	-	100%	-	-	-	-	-
Mushroom	-	10%	-	-	-	80%	-	-	-	10%
Bus	-	-	-	-	-	-	100%	-	-	-
Camera	10%	-	-	-	10%	-	-	80%	-	-
Guitar	-	-	-	-	-	-	-	-	100%	-
Coin	-	-	-	-	10%	-	-	-	20%	70%

Figure 4: Confusion Matrix of the trained CNN using TensorFlow and AutoML Vision.

4 Use of the Cloud Vision API

In other to create an alternative endpoint for image classification that makes use of label detection through the Google Cloud Vision API using the corresponding Python client API, we started by creating in the main a new endpoint similar to the endpoint of the image classify using TensorFlow (see Appendix A). We also created an html template for this endpoint (image_search_multiple.html) similar to the one used for image_search and also added the endpoint in the template index.html. For label detection using the Google Cloud Vision API, we adapted the code taken from the documentation of Google Cloud Vision as can be seen in the following snippet:

```
def detect_labels(file, min_confidence):
    """Detects labels in the file."""
    from google.cloud import vision

    client = vision.ImageAnnotatorClient()

    content = file.read()
    image = vision.Image(content=content)

    response = client.label_detection(image=image)
    labels = response.label_annotations

    labels2 = []
    for label in labels:
        if label.score>=min_confidence:
            labels2.append(dict(description=label.description, score=round(label.score,2)))

    return labels2
```

Our function to detect_labels has as inputs the image file (file) and the minimum confidence (min_confidence), both of them provided by the user of the application, and returns only the labels (and respective score) with the minimum confidence level.

5 Define a Docker image for the app

Regarding the definition of a Docker image for the application, the following code snippet presents the Dockerfile used to build our image. We used a base image from tensorflow, which has already installed Ubuntu 18.04.5 LTS (Bionic Beaver), Python 3.6.9 and TensorFlow 2.4.1. We then updated the pip version (from 20.2.4 to 21.0.1) and installed all the Python packages requirements of the developed application (see Appendix B). In lines 9 and 11 we copied the files required for the application to run as well the Google service account key. In line 18 is set the port number that the container should expose and in lines 20-21 the environment variables GOOGLE_APPLICATION_CREDENTIALS and GOOGLE_CLOUD_PROJECT are also set. In line 23 we then run our application.

```
1  # Base image
2  FROM tensorflow/tensorflow:latest
3  # upgrade pip
4  RUN pip install --upgrade pip
5  # Install Python modules needed by the Python app
6  COPY /app/requirements.txt /usr/src/app/
7  RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt
8  # Copy files required for the app to run
9  COPY /app/. /usr/src/app/
10 # Copy key
11 copy tome-guima-project-a1fa07a3d182.json /usr/src/app/
12 # Tell the port number the container should expose
13 EXPOSE 5000
14 # Set environment variables
15 ENV GOOGLE_APPLICATION_CREDENTIALS="/usr/src/app/tome-guima-project-a1fa07a3d182.json"
16 ENV GOOGLE_CLOUD_PROJECT="tome-guima-project"
17 # Run the application
18 CMD ["python3", "/usr/src/app/main.py"]
```

Finally, it should be mentioned that we had to allocate 512MiB to each container instance since 256MiB were not enough. As already mentioned in the Introduction, our application running in a Docker image can be accessed in https://tome-guima-run-nuxq7zecaa-uc.a.run.app.

6 Conclusions

Our project comprise all proposed objectives, including the additional challenges. The deployment test was made and all the functionalities is working perfectly. During the developed process some constraints have arisen, but with effort and commitment they were overcome. We believe that those obstacles gave us greater ability to understand the entire process of developing the application using the Google Cloud Platform.

Appendixes

A Cloud Vision API endpoint snippets

```python
@app.route('/image_classify_CloudVisionAPI', methods=['POST'])
def image_classify_CloudVisionAPI():
    files = flask.request.files.getlist('files')
    min_confidence = flask.request.form.get('min_confidence', default=0.25, type=float)
    results = []
    if len(files) > 1 or files[0].filename != '':
        for file in files:
            classifications = gcv_detect_labels.detect_labels(file, min_confidence)
            blob = storage.Blob(file.filename, APP_BUCKET)
            blob.upload_from_file(file, blob, content_type=file.mimetype)
            blob.make_public()
            logging.info('image_classify: filename={} blob={} classifications={}'\
                .format(file.filename,blob.name,classifications))
            results.append(dict(bucket=APP_BUCKET,
                                filename=file.filename,
                                classifications=classifications))

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

B Python packages requirements of the developed application

- google-cloud-bigquery==2.12.0
- google-cloud-storage==1.36.2
- google-cloud-vision==2.3.0
- Flask==1.1.2
- tensorflow==2.4.1
- Pillow==8.1.2