Faculdade de Ciência da Universidade do Porto

RELATÓRIO DE BIG DATA & CLOUD COMPUTING (CC4093)

Abril de 2022

up201704877 - Sofia Malpique
up201805440 - Yannik Bauer

Introdução

No âmbito da Unidade Curricular Big Data and Cloud Computing, foi-nos proposto que elaborássemos o presente projecto. Projecto este composto por 2 tarefas de caráter obrigatório e outras 2 tarefas de caráter opcional, chamados de challenges.

Informação Básica acerca do projecto

O nosso projecto foi criado de raíz com o nome bdccProject01 pelos alunos Sofia Malpique e Yannik Bauer.

Project ID: bdccproject01

URL: https://bdccproject01.appspot.com/

< Go Back Start page Project information BDCC homepage

BDCC project app

Endpoint	Description	Test
`/classes`	List image labels.	List
`/image_info`	Get information for a single image.	Image Id: 0041c772f8b9aef6 Get info
`/relations`	List relation types.	List
`/image_search`	Search for images based on a single label.	Description: Cat Image limit: 10 Search
`/image_search_multiple`	Search for images based on multiple labels (**tip:** ARRAY_AGG and UNNEST) may be useful.)	Descriptions (comma-separated): Cat,Dog Image limit: 10 Search
`/relation_search`	Search for images by relation (**tip:** use the LIKE operator).	Class 1 (% for any): % Relation (% for any): plays Class 2 (% for any): % Image limit: 10 Search
`/image_classify_classes`	List available classes for image classification.	List
`/image_classify`	Use TensorFlow model to classify images.	Images: Choose files No file chosen Minimum confidence: 0.05 Classify
`/image_multiple_labels`	Use Google Vision API to detect labels in images.	Images: Choose files No file chosen Classify

Implementação dos Endpoints

Foi-nos pedido que implementássemos os endpoints `/image_info`, `/relations`, `/image_search_multiple`, `/relation_search` e `/image_multiple_labels`.

Este último conta na lista dos desafios, sendo o primeiro de dois.

`/image_info`

`/image_info`	Get information for a single image.	Image Id: 0041c772f8b9aef6 Get info

Para a implementação deste endpoint foi necessário este excerto de código:

```
@app.route('/image_info')
def image_info():
    image_id = flask.request.args.get('image_id')
    imageLabelsResults = BQ_CLIENT.query(
        '''
        SELECT ARRAY_AGG(Description ORDER BY Description) AS ImageDescriptions
        FROM bdccproject01.openimages.image_labels
        JOIN bdccproject01.openimages.classes USING(Label)
        WHERE ImageId = '{0}'
    '''.format(image_id)
    ).result()
    logging.info('image_info: image_id={}, imageLabelsResults={}'.format(
        image_id, imageLabelsResults.total_rows))
    relationsResults = BQ_CLIENT.query(
        '''
        SELECT ARRAY_AGG(Description1) AS fisrtLabel,ARRAY_AGG(Relation) AS possibleRelations,
        ARRAY_AGG(Description) AS Description2
        FROM(
            SELECT ImageId, Description AS Description1, Label1, Relation, Label2
            FROM bdccproject01.openimages.relations
            JOIN bdccproject01.openimages.classes  ON Label = Label1
            )
        JOIN bdccproject01.openimages.classes ON Label = Label2
        WHERE ImageId = '{0}'
    '''.format(image_id)
    ).result()

    data = dict(
        image_id=image_id,
        imageLabelsResults=imageLabelsResults,
        relationsResults=relationsResults
    )

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

- A primeira query é feita à tabela resultante da junção das tabelas image_labels e classes usando a key *Label*.
- O resultado da query é um ARRAY_AGG[1] das descrições das imagens cujo image_id é igual ao id que pesquisamos.
- A segunda query é feita a uma tabela resultante de uma outra query. Query esta que é resultante da junção das tabelas relations e classes onde Label (da tabela classes) é igual a Label1[2] (da tabela relations). A query "de fora" é a junção das tabelas classes e da tabela imediatamente anterior obtida, onde Label2 (da tabela relations) é igual a Label (da tabela classes).
- O resultado da segunda query são 3 ARRAY_AGG que, combinados, contêm todas as relações associadas à imagem em questão.
- Quando estamos no endpoint /image_info, é renderizado o template 'image_info.html', que segue o padrão dos templates dos endpoints que já se encontravam renderizados.

`/relations`

`/relations`	List relation types.	List

Para a implementação deste endpoint foi necessário este excerto de código:

```
@app.route('/relations')
def relations():
    results = BQ_CLIENT.query(
        '''
        Select Relation, COUNT(*) AS NumRelations
        FROM `bdccproject01.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)
```

- Uma query à tabela *relations* onde queremos a relação e a contagem de cada uma.
- Quando estamos no endpoint /relations, é renderizado o template 'relations.html', que segue o padrão dos templates dos endpoints que já se encontravam renderizados.

---

[1] ARRAY_AGG é um array que agrega os resultados.
[2] A tabela relations tinha como atributos, entre outros, Label1 e Label2, onde a composição seria Label1 - Relation - Label2.

`/image_search_multiple`

`/image_search_multiple`	Search for images based on multiple labels.	Descriptions (comma-separated): Cat,Dog Image limit: 10 Search

Para a implementação deste endpoint foi necessário este excerto de código:

```
@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) as GroupedDescriptions
        FROM `bdccproject01.openimages.image_labels`
        JOIN `bdccproject01.openimages.classes` USING(Label)
        WHERE Description IN UNNEST({0})
        GROUP BY ImageId
        HAVING COUNT(Description) >= 1
        ORDER BY COUNT(Description) DESC, ImageId
        LIMIT {1}
    '''.format(descriptions, image_limit)
    ).result()

    logging.info('image_search: 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)
```

- Uma query à tabela resultante da junção das tabelas image_labels e classes, usando a key Label (comum a ambas as tabelas), que contivessem alguma das descrições que constam em UNNEST(array)[3].
- Temos GROUP BY ImageId para que depois se possa renderizar a imagem e listar quais as imagens que têm como label pelo menos uma das descrições pesquisadas.
- Quando estamos no endpoint /image_search_multiple, é renderizado o template ‘image_search_multiple.html’, que segue o padrão dos templates dos endpoints que já se encontravam renderizados.

[3] UNNEST recebe um array e retorna uma tabela com um elemento por linha.

`/relation_search`

`/relation_search`	Search for images by relation.	Class 1 (% for any): % Relation (% for any): plays Class 2 (% for any): % Image limit: 10 Search

Para a implementação deste endpoint foi necessário este excerto de código:

```
@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 ImageId
        FROM(
            SELECT ImageId, Description, Relation, Label2
            FROM `bdccproject01.openimages.relations`
            JOIN `bdccproject01.openimages.classes` ON Label = Label1
            WHERE Description LIKE '{0}' AND Relation LIKE '{2}'
            )
        JOIN bdccproject01.openimages.classes ON Label = Label2
        WHERE classes.Description LIKE '{1}'
        ORDER BY ImageId
        LIMIT {3}
    '''.format(class1, class2, relation, 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('relations_search.html', data=data)
```

- A query é feita a uma tabela resultante de uma outra query. Query esta que é resultante da junção das tabelas relations e classes onde Label (da tabela classes) é igual a Label1 (da tabela relations). A query “de fora” é a junção das tabelas descriptions e da tabela imediatamente anterior obtida, onde Label (da tabela classes) é igual a Label2 (da tabela relations).
- Usamos o operador LIKE para que possamos usar ‘%’ para representar qualquer relação ou qualquer label. É usado o carater ‘%’ porque em SQL trata-se de um carater especial que representa “any”.
- Quando estamos no endpoint /relation_search, é renderizado o template ‘relation_search.html’, que segue o padrão dos templates dos endpoints que já se encontravam renderizados.

`/image_multiple_labels`.

`/image_multiple_labels`	Use Google Vision API to detect labels in images.	Images: Choose files No file chosen Classify

Para a implementação deste endpoint foi necessário este excerto de código:

```python
@app.route('/image_multiple_labels', methods=['POST'])
def image_multiple_labels():
    files = flask.request.files.getlist('files')
    credentials = service_account.Credentials.from_service_account_file('app/key.json')

    to_show = []

    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()
            logging.info('image_multiple_labels: filename={} blob={}'
                         .format(file.filename, blob.name))

            image_uri = 'gs://bdccproject01.appspot.com/' + file.filename

            client = vision.ImageAnnotatorClient(credentials=credentials)
            image = vision.Image()
            image.source.image_uri = image_uri

            response = client.label_detection(image=image)  # pylint: disable=no-member
            results = []
            results.append(dict(bucket=APP_BUCKET,
                                filename=file.filename))

            for label in response.label_annotations:
                results.append((label.description,label.score*100))

            to_show.append(results)

    data = dict(bucket_name=APP_BUCKET.name,
                results=to_show)

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

- Neste endpoint foram necessárias credenciais para autenticação.
- Usamos a API do Google Cloud Vision para, dada uma imagem, listar as diferentes labels possiveis da imagem.
- O código foi ajustado com o auxílio dos scripts disponibilizados pelos tutoriais do Google Cloud Platform.
- Este endpoint também segue os padrões dos templates dos outros endpoints.

Script de preparação do dataset do TensorFlow

Devemos correr o script create_dataset_csv.py para que o dataset com destino ao modelo TensorFlow seja criado (se ainda não foi) e criar no Google Cloud Storage um bucket auxiliar que receberá as 1000 imagens e o ficheiro .csv oriundo do script.

Primeiramente abre-se o ficheiro dict.txt que contém uma lista de 10 classes possíveis para as imagens a classificar.

Depois, para poupar alguma redundência definiu-se uma função generateQuery, que recebe como parâmetro uma categoria e retorna uma string, cujo conteúdo é uma query que retornará 100 ImageIds de imagens que tenham apenas aquela categoria como label.

Para cada uma das categorias, elaboramos uma query que retornará a informação necessária (ImageId e categoria/label). Com o resultado da query elaboramos dois ficheiros distintos: ImagesURIs.txt e autoMLvision.csv.

O primeiro ficheiro, ImagesURIs.txt, contém 1000 linhas do tipo gs://bdcc_open_images_dataset/images/<ImageId>. O segundo ficheiro contém também 1000 linhas em que se dividem em 10 categorias diferentes, com 100 imagens cada categoria e cada grupo de 100 imagens tem uma distribuição de 80 para treino, 10 para validação e 10 para teste.

Depois de tudo isto, devemos fazer upload das 1000 imagens para o bucket auxiliar e o ficheiro .csv acima descrito.

Criamos um dataset no UI do AutoML do Google Cloud e lá fazemos upload ao ficheiro .csv para que possa a seguir treinar.

O processo de treino demora algumas horas.

Depois faz-se deploy ao modelo TensorFlow Lite e extraiem-se ficheiros como model.tflite, que colocamos em app/static/tflite.

Dockerfile

O nosso Dockerfile é o seguinte:

Dockerfile

```
1   FROM python:3.8
2
3   ENV GOOGLE_CLOUD_PROJECT bdccproject01
4
5   WORKDIR /firstdocker
6
7   COPY ./app/requirements.txt .
8
9   RUN pip3 install -r requirements.txt
10
11  COPY ./app ./app
12
13  CMD ["python", "./app/main.py"]
14
```

FROM - Especificação da imagem base para as instruções subsequentes. Estamos a estender a imagem dos repositórios públicos do docker para ser utilizada como boiler plate neste Dockerfile.

ENV - Estamos a definir a varíavel de ambiente GOOGLE_CLOUD_PROJECT dentro do docker.

WORKDIR - Definição do dirétório principal.

COPY - Estamos a copiar o ficheiro requirements.txt para a root do docker.

RUN - Estamos a dar a instrução para instalar as dependências do Pyhton que necessitamos.

COPY - Estamos a copiar toda a pasta local app para uma pasta no docker, também chamada app.

CMD - Estamos a dar a intrução para executar o ficheiro main.py com o Python (versão 3).