

Aplicação com *App Engine* e recurso a *TensorFlow* e *AutoML Vision*

Rui Ramos up201705782

Rúben Lôpo up201709326

Abril 2021

Departamento de Ciência de Computadores

Big Data & Cloud Computing



Conteúdo

| | | |
|----------|--|-----------|
| 1 | Introdução | 3 |
| 2 | Programação dos <i>app endpoints</i> | 3 |
| 2.1 | <i>image_info endpoint</i> | 3 |
| 2.2 | <i>relations endpoint</i> | 5 |
| 2.3 | <i>relations_search endpoint</i> | 5 |
| 2.4 | <i>image_search_multiple endpoint</i> | 6 |
| 3 | Criação de um modelo <i>TensorFlow</i> usando <i>AutoML</i> | 8 |
| 3.1 | Conexão, inicialização de <i>Dataframes</i> e definição de classes . . . | 8 |
| 3.2 | Definição do <i>Dataset</i> | 9 |
| 3.3 | Alocação dos dados num <i>bucket</i> | 10 |
| 3.4 | Criação, treino e aplicação dos modelos no <i>AutoML Vision</i> . . . | 11 |
| 4 | Conclusão | 12 |

1 Introdução

No âmbito da unidade curricular de *Big Data & Cloud Computing*, este projeto incidiu na vertente de programação *Cloud* e permitiu a programação de uma aplicação usando *App Engine*, com o objetivo de mostrar informação acerca de imagens alocadas num *Dataset*.

Além disto, foi desenvolvido um serviço através de um modelo *TensorFlow* para classificação de imagens usando *AutoML Vision*.

Numa fase inicial, foram programados os endpoints que estavam em falta, através de consultas de data em *BigQuery*, que acede às imagens num *bucket* público.

Numa segunda fase, criou-se um modelo *TensorFlow* com *AutoML* próprio e substituiu-se o que foi dado. O *dataset* usado durante este processo foi criado com recurso a *Apache Spark* num ficheiro editável do *Google Colab*.

O identificador do *Google Cloud Project* é **cloud-computing-project-309514**.

O URL da aplicação *App Engine* é

<https://cloud-computing-project-309514.uc.r.appspot.com>.

2 Programação dos *app endpoints*

A pasta da aplicação está organizada, essencialmente, através de **ficheiros html** e um **ficheiro python main**.

Cada *endpoint* efetua uma (ou mais) consultas *BigQuery* e envia a informação resultante, juntamente com outras necessárias, para a construção da página html com o mesmo nome.

As páginas html estão adaptadas de forma a que recebam o resultado das consultas e as apresentem de uma forma legível para quem efetua a pesquisa.

Os *endpoints* **classes**, **image_search**, **image_classify_classes**, **image_classify** serviram de modelo à criação de outros endpoints.

O ficheiro **index.html** contém todo o html do menu inicial, bem como os botões configurados para permitir ao utilizador especificar valores e parâmetros usados nas consultas.

2.1 *image_info endpoint*

Neste *endpoint*, o utilizador fornece o *ImageId* da imagem que quer consultar e esta informação é recolhida através de:

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

Como apenas existe um argumento recebido, este é indexado com zero.

Para a resolução deste *endpoint*, foram realizadas **duas consultas distintas**, que possuem em comum o mesmo *ImageId*.

A **primeira consulta retorna todas as classes da ImageId específica**, dada anteriormente no **index.html**, através da seguinte consulta:

```

    results1 = BQ_CLIENT.query(
'''
    Select Description, ImageId
    FROM 'bdcc21project.openimages.image_labels'
    JOIN 'bdcc21project.openimages.classes' USING(Label)
    WHERE ImageId = '{0}'
    ORDER BY Description

'''.format(image_id)
    ).result()
    logging.info('image_info: image_id={} results1={}'.format(
        image_id, results1.total_rows))
    data1 = dict(image_id=image_id,
                 results1=results1)

```

Esta consulta utiliza as tabelas image_labels e classes, através de um join usando o coluna Label, e procura as Descriptions das Labels das classes da imagem com ImageId especificado.

A **segunda consulta retorna as relações presentes na imagem requisitada** através do seguinte código:

```

results2 = BQ_CLIENT.query(
'''
    SELECT DISTINCT
        (SELECT Description FROM bdcc21project.openimages.
classes WHERE Label=Label1),
        Relation,
        (SELECT Description FROM bdcc21project.openimages.
classes WHERE Label=Label2),
        ImageId
    FROM bdcc21project.openimages.image_labels
    JOIN bdcc21project.openimages.relations USING(ImageId)
    WHERE ImageId = '{0}'

'''.format(image_id)
    ).result()
    logging.info('image_info: image_id={} results2={}'.format(
        image_id, results2.total_rows))
    data2 = dict(image_id=image_id,
                 results2=results2)

```

A consulta seleciona os elementos distintos de um join das tabelas image_labels e relations através da Description da Label1, da sua Relation, da Description da Label2 e ImageId, para a ImageId específica.

São retornados os seguintes argumentos, para a página html, de ambas as consultas.

```

return flask.render_template('image_info.html', data1=data1, id
=id, data2=data2)

```

2.2 relations endpoint

Neste *endpoint*, o objetivo passou por **listar o número de *Relations* que existem entre imagens do dataset**. Para tal, efetuou-se a seguinte consulta:

```
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)
```

Nesta consulta seleciona-se a *Relation* e contabiliza-se o número de imagens com essa relação, na tabela *relations*, ordenadas por ordem alfabética.

Retorna-se apenas o resultado da consulta para o html com:

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

2.3 relations_search endpoint

Para o *endpoint* *relations_search*, o utilizador procura por um conjunto de *Class1* / *Relation* / *Class2*, em que as classes podem ser específicas ou default (denotadas com %), além de se especificar o limite de imagens a apresentar no html. Portanto, são recebidos os argumentos em python da seguinte maneira:

```
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)
```

A consulta é realizada da seguinte forma:

```
results = BQ_CLIENT.query(
'''
    SELECT ImageId,
    (SELECT Description FROM bdcc21project.openimages.
classes WHERE Label=Label1) AS Description1,
    Relation,
    (SELECT Description FROM bdcc21project.openimages.
classes WHERE Label=Label2) AS Description2,
    FROM bdcc21project.openimages.relations
    WHERE (SELECT Description FROM bdcc21project.openimages.
classes WHERE Label=Label1) LIKE '{0}' AND Relation LIKE '{1}'
    AND (SELECT Description FROM bdcc21project.openimages.classes
WHERE Label=Label2) LIKE '{2}'
    ORDER BY ImageId
''')
```

```

LIMIT {3}
''' .format(class1, relation, class2, image_limit)
).result()
logging.info('relation_search: class1={} relation={} class2={}
image_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)

```

Nesta consulta são selecionadas a ImageId, a Description da primeira Label, a Relation e a Description da segunda Label. Para isto, faz-se a consulta à tabela relations de forma a que os atributos selecionados pelo utilizador sejam utilizados. É importante referir que foram feitos dois SELECT dentro do SELECT principal, de forma a obter a Label a partir da Description introduzida pelo utilizador. O limite também é dado pelo utilizador e é usado para limitar a consulta.

Todos estes atributos estão denotados pela ordem que são passados à consulta, sendo a Class 1 indexada por 0, a Relation indexada por 1, a Class 2 indexada por 2 e o Image limit por 3.

São passados ao html tanto o resultado da consulta como os inputs do utilizador (para formatar corretamente a página html), além do total de resultados encontrados.

```

return flask.render_template('relation_search.html', class1=
class1, relation=relation, class2=class2, image_limit=
image_limit, data = data, total = results.total_rows)

```

2.4 image_search_multiple endpoint

O objetivo desta consulta é procurar por imagens com uma ou mais classes dadas como input pelo utilizador.

São passados como inputs o limite de imagens a apresentar e um *array* descriptions com todas as classes especificadas.

```

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

```

Segue-se a consulta efetuada:

```
results = BQ_CLIENT.query(
'''
    SELECT ImageId, ARRAY_AGG(Description), COUNT(Description)
    FROM bdcc21project.openimages.image_labels
    JOIN bdcc21project.openimages.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)
```

Nesta consulta são selecionadas todas as imagens que correspondem a pelo menos uma classe dada pelo utilizador, assim como a quantidade de classes que correspondem a essa consulta. Esta procura é feita na tabela `image_labels` juntamente com a tabela `classes`, através da coluna `Label`.

Para isto, utilizam-se duas funções distintas:

- `ARRAY_AGG(Description)` foi utilizada para retornar, na consulta, um *array* com as classes associadas ao `ImageId` em questão.
- `UNNEST ({0})` foi utilizada para transformar o *array* com as classes, dadas como input, numa tabela, de forma a que se pudesse verificar se uma descrição estava contida nos valores dessa mesma tabela.

No fim, são enviados para o htm os dados resultantes, o array `descriptions`, o limite de imagens a ser visualizado e o total de resultados obtidos na consulta, bem como o número de descrições dadas como input pelo utilizador.

```
return flask.render_template('image_search_multiple.html', data
=data, descriptions=descriptions, image_limit=image_limit,
total = results.total_rows, len_descriptions = len(descriptions
))
```

3 Criação de um modelo *TensorFlow* usando *AutoML*

3.1 Conexão, inicialização de *Dataframes* e definição de classes

Este processo foi dividido em várias etapas, usando o esqueleto fornecido no Google Colab:

- Em primeiro lugar, deu-se **setup** ao *Spark*.
- Em seguida, fez-se a **conexão** à *Google Cloud*, inserindo o ID do projeto e o URI do *Bucket*.

```
PROJECT_ID = 'cloud-computing-project-309514'
BUCKET_URI = 'gs://bdcc_open_images_dataset'
from google.colab import auth
auth.authenticate_user()
!gcloud config set project {PROJECT_ID}
```

- Em terceiro lugar, **obteve-se** os dados necessários relativos às classes e imagens a utilizar nas tabelas *BigQuery* do projeto e inicializou-se os *dataframes* com os mesmos dados.
- Em último lugar, **definimos** as 10 classes escolhidas para a tarefa de **classificação**.

```
CLASSES =[
    ('Squirrel',),
    ('Flag',),
    ('Coin',),
    ('Ball',),
    ('Falcon',),
    ('Glove',),
    ('Goat',),
    ('Taco',),
    ('Computer monitor',),
    ('Knife',)
]

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


3.2 Definição do *Dataset*

Depois de configurado o *Spark* e escolhidas as classes, a tarefa seguinte envolveu a criação de um *dataset* com uma formatação própria para ser utilizado no *AutoML*. Segue-se o código de criação do *dataset* e a posterior explicação do mesmo.

```
import pandas as pd
dataToCSV = []
listCSVToImagens = []

for classAux in CLASSES:
    query = spark.sql('''
        SELECT * FROM image_labels
        JOIN classes USING(Label)
        WHERE Description = '{0}'
        LIMIT 100
    '''.format(classAux[0]))
    numbersOfMLControl = 0
    for row in query.rdd.collect():

        if numbersOfMLControl < 80:
            typetoCSV = "TRAIN"
            uritoCSV = "gs://projectbucket10/images/" + row.ImageId +
            ".jpg"
            classToCSV = row.Description
        if numbersOfMLControl >= 80 and numbersOfMLControl < 90:
            typetoCSV = "VALIDATION"
            uritoCSV = "gs://projectbucket10/images/" + row.ImageId +
            ".jpg"
            classToCSV = row.Description
        if numbersOfMLControl >= 90:
            typetoCSV = "TEST"
            uritoCSV = "gs://projectbucket10/images/" + row.ImageId +
            ".jpg"
            classToCSV = row.Description

        numbersOfMLControl = numbersOfMLControl + 1

        csvLine = []
        csvLine.append(typetoCSV)
        csvLine.append(uritoCSV)
        csvLine.append(classToCSV)
        dataToCSV.append(csvLine)

        uritoList = "bdcc_open_images_dataset/images/" + row.
        ImageId + ".jpg"
        listCSVToImagens.append(uritoList)

    numbersOfMLControl = 0

# Create the pandas DataFrame
df = pd.DataFrame(dataToCSV)
csv = df.to_csv(index=False, header=False)
file = open('csvClasses.csv', mode='w')
file.write(csv)
file.close()
```

O procedimento envolveu as seguintes fases:

- Inicialmente, para **cada Classe** que foi definida anteriormente realizou-se uma consulta de forma a selecionar 100 entradas para cada uma.
- Seguidamente, **iterou-se todas as entradas**, de cada conjunto de 100, dividindo-as em 80 para treino, 10 para validação e 10 para teste. Além disto, extraiu-se a ImageId e a Description, sendo estas as 3 informações que devem constar no CSV. Guardou-se, também, o caminho de cada imagem no bucket `bdcc_open_images_dataset` para, posteriormente, se conseguir efetuar uma cópia de cada imagem para o nosso bucket.
- Por fim, utilizando os dados extraídos, **criou-se um *dataframe*** que foi convertido em CSV e guardado nos ficheiros do nosso projeto.

3.3 Alocação dos dados num *bucket*

Esta secção, após a criação de um *bucket* dedicado ao processo do AutoML, envolveu dois passos:

- Em primeiro lugar, **copiou-se o CSV** que foi criado anteriormente, dos ficheiros do nosso projeto para o *bucket* `projectbucket10`.

```
MY_AUTOML_BUCKET='projectbucket10'
!gsutil -m cp -R /content/csvClasses.csv gs://{
MY_AUTOML_BUCKET}
```

- Em segundo lugar, **foram copiadas todas as imagens selecionadas para a tarefa de classificação**, através dos caminhos anteriormente guardados na variável `listCSVToImagens`, referentes ao bucket `bdcc_open_images_dataset`.

```
MY_AUTOML_IMAGES = 'projectbucket10/images/'

for img in listCSVToImagens:
    !gsutil -m cp -R gs://{img} gs://{MY_AUTOML_IMAGES}
```

3.4 Criação, treino e aplicação dos modelos no *AutoML Vision*

Numa fase final, foi utilizado o *AutoML Vision* e foram criados vários modelos de classificação, onde se escolheu o que reunia melhores resultados. Seguem-se algumas das informações obtidas na plataforma, relativamente à qualidade do modelo:



4 Conclusão

Em conclusão, este trabalho permitiu adquirir conhecimentos relativos a programação *Cloud*, com conexões a *datasets* e consultas a esses mesmos *datasets* de forma a tornar possível, através de uma aplicação, aceder à informação contida nesse espaço.

Concluído o trabalho, o **grupo foi capaz de acabar todos os *endpoints* e criar a classificação de imagens com o seu próprio modelo *TensorFlow* com *AutoMLVision*, tornando todas as funcionalidades disponíveis.**

Em suma, consideramos que este trabalho foi uma mais valia e que os conhecimentos angariados foram de extrema utilidade, além dos conceitos intrigantes e desafiantes que proporcionou.