Numberfigure program: using mnist online

Qianqian Cui

Contents

1 Abstract **1**

2 Introduction **3**
2.1 Related Background 3
2.2 Dependency 3
2.2.1 Python 3
2.2.2 Cassandra 4
2.2.3 JavaScript 5
2.2.4 Docker 5

3 Procedure **5**
3.1 File Submitting 5
3.1.1 Build the website framework 5
3.1.2 Set file submit function and return file 6
3.1.3 Get file and use MNIST model to predict 6
3.1.4 Cassandra upload 7
3.1.5 Return the output 8
3.2 Draw-board Submitting 8
3.2.1 Return json file to Python 10
3.2.2 Uncode file and predict 11
3.2.3 Cassandra and Return 12

4 Result **13**
4.1 Result for file submitting 13
4.2 Result for draw-board submitting 15

5 Concousion **17**

6 References and Links **17**

1 Abstract

This report is about a program, whose main work is to use MNIST model to figure out the number images of different input on a website, and upload related data up to the Cassandra database. The program is based on Python3, Jquery, Cassandra and Docker. By using trained model, the user could input a number either by submitting a number image or by drawing a number on a given draw-board. The whole process can be operated on Docker container.

Keywords: MNIST, Json, Cassandra, Docker

2 Introduction

2.1 Related Background

MNIST is a basic and well-known deep learning model, which is usually realized by Python. It has been created and improved by many developers. The model's main function is to figure out the number in the hand-written number image, after being trained and tested by the given image database. Usually, if we need an effective MNIST model, we can get the well-developed code on Github**[2]** first, download example training sets and testing sets**[2]**, then use restore function in Python to get one. To imply the model on real hand-written images, we can set the image as an input of the model directly.

However, this method is not operable enough, especially when a user need to write a number immediately and upload it. Moreover, the interface of the python graphical operation is not friendly enough to the beginners.

This program offers a better strategy to use the given MNIST model**[3]**. Based on Python-flask web framework, JavaScript, Cassandra database and Docker container, the program satisfied the function, that through web server, the user could upload number image or draw a number directly on the website, then he/she could get the relative prediction of the image from MNIST model. Furthermore, the program offers to upload the process information onto a certain Cassandra table during the uploading. The whole program could either be used on local computer or placed in docker successfully.

Thanks to Dr.Zhang, who introduces me to a deeper learning of not only a series of related software and database, but the fundamental structure of the big-data industry, I can integrate those multiple objects to generate this program.

2.2 Dependency

In this program, the dependent language modules/database/software and their details is list as below.

2.2.1 Python

This program requires Python whose version is over 3.5 or more(Attention: since up till 2019-08, the tensorflow module is not ready for python 3.7, please take care if you're using 3.7version). The following modules needed is separated by their functions. Here I only list the major modules.

```
1.Flask modules. The useful function is as below:
```

[16]:
```
from flask import Flask, render_template, request, jsonify, abort
```

Here Flask is to build the basic web server using flask; render_template is to return the object in the Python function to a certain html website; request is to inspect the type of the 'requests' from html and to do further process; jsonify offers a way for the Python function to return a data in json form to the html; abort, which works similarly in html, offers an alert message in html pages.

```
2.Image Processing modules. The useful function is as below:
```

[17]:
```
from PIL import Image
import matplotlib.pyplot as mt
import io
from urllib.parse import unquote
import base64
```

The PIL module in Python3 is actually Pillow, and by using it with 'pyplot' module, we can do simple process on the input image, to change it into the necessary form for the MNIST model.

The io, urllib and base64 modules are mainly for images return from the online draw-board. They can manage to turn a json file to a base64 form image, then to a PIL form image.

```
3. Cassandra-driver module. The useful function is as below:
```

[18]:
```
from cassandra.cluster import Cluster
```

This module is responsible for connecting Cassandra and Python commands. Set right ports and ip before connecting the two servers.

```
âĔč. Tensorflow module. The useful function is as below:
```

[19]:
```
import tensorflow as tf
```

To check its main function, please go for the link**[1]**.

2.2.2 Cassandra

You can use the latest version of Cassandra database to serve for the program. Personally, I use docker to build a Cassandra server:

[20]:
```
# ~$ docker pull cassandra
```

After building Cassandra server, you can use the following command to make a container for Cassandra data storage:

[21]:
```
# ~$ docker run -d --name [containername] -p [port] cassandra:latest
```

Here we use the given cassandra image from docker. When you want to check the information in this container, you can use:

[22]:
```
# ~$ docker exec -it [containername] cqlsh
```

To run this program successfully, you should build a certain keyspace and a certain table to keep the connection from Python work. You can build them either in Cassandra itself or in Python**(the program offers a funciton 'createKeySpace()' to build them in csd.py)**.

2.2.3 JavaScript

Since this program is relative to the interaction of html and background Python program, JS is indispensable to build interaction and other functions. Although html itself need nothing but a modern web browser to realize, the Jquery part in JS still needs related environment.

To run this program, you could use either the static json file in **'root/flaskandmnist/static/' path(and it's the program's default method)** or use the official link to use the ajax service.

2.2.4 Docker

Docker can not only save the Cassandra database, but save the whole program onto an image(and certain container) . To set the program onto Docker, you can use the Dockerfile to help upload the image. To see an example uploading process, check **[4]**. Moreover, if you use docker to run the program, you should make sure the port of the 'flask_1.py' is equal to the one of the target container, and keep the network of the two container(program and cassandra) as the same.

3 Procedure

The program has two functions separately. The first function is returning the result by submitting an image file, and the second one is returning the result by writing a number on a given draw-board and submitting. So the procedure would be two parts, explaining the two method in turns.

3.1 File Submitting

The description of this function is that: it lets user submit an number image by clicking the button on the given website, and it returns the prediction of MNIST model towards this image, while uploading the image information onto Cassandra. The main procedures is listed as below.

3.1.1 Build the website framework

The first step is made up of simple codes.

```
[23]: # 1.1 set app
      app = Flask(__name__)
      # 1.2 set index page
      @app.route('/')
      def getpic():
          return render_template('index.html')
```

Here, we use Flask(**name**) to set default dictionary. Use render_template to set 'index.html' as the beginning page.

3.1.2 Set file submit function and return file

In the former step, the return of the 'gitpic()' function turn the page to load as 'index.html'. Here, to create a submit button, which function is to give out a request as 'post', and to shift to the output page, I write the code in 'index.html', as below:

```
[ ]: # // code 0.1
     # <form method = 'POST' enctype = multipart/form-data action = "http://localhost:
      ↪5000/upload">
     #     <input type = "file"  name = "file" />
     #     <input type = "submit" />
     # </form>
```

Such form offers to return the file with a variable called 'file' to Python-flask. Also, it serves as a link to **'/upload'**.

3.1.3 Get file and use MNIST model to predict

In the previous step, our server now turns to '/upload' page. Here, I write another function called 'processpic()' to receive 'file', analyze it and make an output. Here is the general code of the function.

```
[29]: # 1.3 set answer page no.1
      # by submitting required pictures from index.html, we get answer using the␣
       ↪following function
      @app.route('/upload', methods = ['GET','POST'])
      def processpic():
          if request.method == 'POST':
              # get image with filestorage form
              pic = request.files['file']
              # get name of the input image
              picname = pic.filename
              # save the image in wd path
              path0 = os.path.dirname(__file__)
              path1 = os.path.join(path0,'saveimage',picname)
              pic.save(path1)
```

```
        # analyze the image with tensorflow model
        pic1 = Image.open(path1)
        target = MNIST_r.get_result(pic1,picname)
        #  send the message to another docker container
        csd.insertnewinfo(target)
    else:
        target = None
    return render_template('output.html',target = target[1])
```

To make the steps clearer, I would explain the 'get and predict' content here only. The 'methods' option in **line[3,5]** is to limit the type of request port. I use **request.files[targetvariable]** here to get the image data from html. After getting the basic information of the image and saving it at a relative path in word dictionary, I invoke the function **in line[16] 'get_result(picture, filename)'** so that the image could be analyzed by the MNIST model.

Here is the get_result function. It's in MNIST_r.py. Since the original model only offers the training and testing process, I trained the model beforehand(20000 times, accuracy 99.41%) and add some details to create this function.

[30]:
```
# side code 1
def get_result(pic,filename):
  result = importpic(pic)
  # Create the model
  x = tf.placeholder(tf.float32, [None, 784])
  y_conv, keep_prob = deepnn(x)
  # def the saver of the model
  saver = tf.train.Saver()
  with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    # get work dictionary path and read the model exec by [link1]
    pathhalf = os.path.dirname(__file__)
    pathwhole = os.path.join(pathhalf, 'mnist_model2.ckpt')
    saver.restore(sess,pathwhole)
    # predict the result number
    predict = tf.argmax(y_conv,1)
    predint = predict.eval(feed_dict = {x:[result],keep_prob:1.0}, session =␣
 ↪sess)
    # print(predint)
    # print("the prediction result is : %d" %predint[0])
    predintt = str(predint[0])
    result_m = (filename,predintt,time_setting())
  return result_m
```

Briefly speaking, this function reloads the completed model to analyze the image('result')**[line12,14]**, and returns a tuple with imagename, prediction result and the present time in it.**[line19,21]** These data would be used both in output page and in Cassandra upload process.

3.1.4 Cassandra upload

The uploading process is also combined in the processpic() function **In [29], line[4]**. I use a function called 'insertnewinfo(info)' to realize it. It's in csd.py.

Here is the general code of this function.

[31]:
```
# side code 2
KEYSPACE = "uploadinfo2"
def insertnewinfo(target):
    cluster = Cluster(contact_points=['127.0.0.1'], port=9042)  # set ip and port
    session = cluster.connect()  # connection
    try:
        log.info("setting keyspace...")
        session.set_keyspace(KEYSPACE)
        log.info("inserting into table...")
        session.execute(
            # here the 'info1' as below is the name of the target table, if␣
 ↪needed, change its name.
            """
                INSERT INTO info1(updatetime, filename, output)
                VALUES(%s, %s, %s)
            """,
            (target[2], target[0], target[1])
        )
    except Exception as e:
        log.error("Unable to upload, please check the code.")
        log.error(e)
```

Similar to creating a keyspace/table in Cassandra, here we could transfer the information by berely using 'INSERT INTO; VALUES' command **line[12,15]**. Please notice that the target Cassandra container has the port '9042', and the target keyspace/table have their corresponding names **(notice that here the table is called 'info1'[13])**.

3.1.5 Return the output

In the processpic() function, it finally returns a html called 'output.html' **In [29],line[21]**, and send a str variable called 'target'**(according to the get_result function, this is the prediction of the model)** onto this html page. We could see the output on this page by quoting this variable.

[ ]:
```
# // code 0.2
#     <br> <font size ="6">{{ target }} </font>
```

To see the whole process on website page, please go to Result part.

3.2 Draw-board Submitting

Based on the work in 2.1, let's build the draw-board function. Since some part of the process is similar to that in 2.1, I would skip it later in the explanation. ### Build draw-board in html To get into the draw-board page, I used a link in 'index.html', it's like:

[ ]:
```
# // code 0.3
#       <form action = "http://localhost:5000/upload2">
#           <button type="submit">Continue</button>
#       </form>
```

After getting into the '/upload2' page, I used the same strategy in 2.1.1 to shift the page to the draw-board **(jsonpic2.html)**.

[35]:
```
# 1.4 set drawboard page
# by clicking the shift link we could get here.
# use json script to create the drawboard and use ajax to return data to '/
 ↪upload3'
@app.route('/upload2', methods = ['GET','POST'])
def movetojs():
    return render_template('jsonpic2.html')
```

Then the main part comes: I created a draw-board using canvas. Since the length of the code, I would only show the script part. The whole code is in **'flaskandmnist/templates/jsonpic2.html'**.

[ ]:
```
# // code 0.4
# <script type = "text/javascript">
#      //  get canvas settings
#      let canvas = document.getElementById('canvas');
#      const ctx = canvas.getContext('2d');
#      //  set original positions
#      let startP = {x: 0, y: 0};
#      let endP = {x: 0, y: 0};
#          // set the background as white and set the position of the drawboard
#      ctx.fillStyle = "rgb(255,255,255)";
#      ctx.fillRect(0,0,500,500);
#      //  add MOUSEDOWN: start writing ; add MOUSEUP: end writing
#      canvas.addEventListener('mousedown', mousedown);
#      canvas.addEventListener('mouseup', mouseup);

#      function mousedown(e) {
#          //  the start point is the mouse position
#          startP = {x:e.clientX,y:e.clientY};
#          //  focus on the start point as beginning
#          ctx.moveTo(startP.x, startP.y);
#          console.log('Mouse down.');
#          canvas.addEventListener('mousemove', mousemove);}
#      function mouseup(e) {
```

```
#         console.log('Mouse up.');
#         canvas.removeEventListener('mousemove', mousemove);}
#         function mouseout(e) {
#                 console.log('Mouse out.');
#                 canvas.removeEventListener('mousemove', mousemove);}

#     // make continuous lines
#     function mousemove(e) {
#         ctx.beginPath();
#         ctx.moveTo(startP.x, startP.y);
#         //  set end position of every step
#         endP = {x:e.clientX,y:e.clientY};
#         console.log(JSON.stringify(startP) + ',' + JSON.stringify(endP));
#         //  focus on the end position of steps
#         ctx.lineTo(endP.x, endP.y);
#         //  print on the position
#         ctx.stroke();
#         // set the thickness of the painting pen
#         ctx.lineWidth = 40;
#         //  get another step(every point as a step to connect to a line)
#         startP = endP;
#         ctx.moveTo(startP.x, startP.y);}
# // set drawboard over
# </script>
```

To check the function of every step, we can check the comments on it. This script offers a 500*500 simple draw-board. We can draw on it using black pen.

3.2.1 Return json file to Python

After building the draw-board, it's necessary to save the canvas and return it to Python-flask for prediction. I used ajax to make the interaction between flask and html here. Also, only part of code(and comment) is shown here.

[ ]:
```
# // code 0.5
# // the ajax works for two process:
# // 1. return a json string from canvas drawboard to flask
# // 2. get the status from flask and turn to another page(/result) with data
#     let save = $('#save');
#     save.on('click', function() {
#         let picc = canvas.toDataURL("image/jpeg",1.0);
#         let data = {filedata: picc};
#                 $.ajax({
#                         type: 'POST',
#                         url: '/upload3',
#                         data: data,
#                          dataType: 'json',
#                          contentType: "application/json;charset=UTF-8",
#                          async:false,
#                          success: function(data){
#                                  // here is for the second process:
#                  if (data.status === 'success') {
#                      console.log(data);
#                      let result = $('#result');
#                      window.location.href = `/result?data=${data.data}`;}},
#                          error: function(data) {console.log('error')}
#                  });
#       })
```

As is said in the comment**line[2,4]**, the ajax works for two process. The two process works by clicking the button whose id is 'save'.

Due to the property of canvas, it could only return certain forms of data. Here I use '.tuDataURL' **line[7]** to return the dataurl string to ajax, and packaged it as json to return by POST.

To let Python-flask notice the POST request from ajax, I used the same strategy as in 2.1.3.

3.2.2 Uncode file and predict

After Python-flask get the json data, it should recognize the data in correct form, change it into needed form and make the prediction. I used jsonpic() function to realize these.

[37]:
```
# code 1.5 set processing page
# since the canvas drawboard returns a json file with base64 code,
# here we split the target code and decode it,
# then we use the similar process as that in code 1.3
@app.route('/upload3', methods=['POST'])
def jsonpic():
    if request.method == 'POST':
        # get the data from ajax
        data = request.get_data().decode('utf-8')
        data = unquote(data)
        # transform data into PIL image
        img_base64 = data.split('=')[1]
        # print(img_base64)
        datapic = transfile.produceImage(img_base64)
        pic2 = Image.open(datapic[0])
        picname2 = datapic[1]
        target2 = MNIST_r.get_result(pic2,picname2)
        csd.insertnewinfo(target2)
        return jsonify(status='success', data=target2[1])
        # here data is sent back to ajax; and ajax saves the data while getting␣
 ↪to the '/result' page
```

Apart from the similar process in 2.1, here we have additional part to handle with the json file and change it into correct form. I used 'unquote', 'split' and function 'produceImage(img64)' to realize it. The produceimage function is in transfile.py.

[38]:
```python
# side code3
def base64toImage(file_in):
    # the b64decode() function asks '==' as its endding, but the return value␣
↪from json doesn't give it.
    # if your return value from json does have such endding, you can pass this␣
↪line
    # (you can check the base64 code by uncommenting line76 in flask_1)
    file_in = file_in.split(",")[1] + "==" #
    file_in = base64.b64decode(file_in)
    # into PIL image
    file_in = io.BytesIO(file_in)
    image = Image.open(file_in)
    return image
# this function is to change the size of image into 28*28px, and then make the␣
↪binarization.
def produceImage(txt):
    image = base64toImage(txt)
    # change the size to 28*28
    resized_image = image.resize((28, 28), Image.ANTIALIAS)
    #  get gray-scale image first
    image2 = resized_image.convert('L')
    # Binatization part
    table = []
    for i in range(256):
        if i < 100:
            table.append(0)
        else:
            table.append(1)
    image2 = image2.point(table,'1')
    # save the processed image by a relative path
    now = time.strftime("%Y-%m-%d-%H_%M_%S", time.localtime(time.time()))
    file_outname = now + r"image.jpg"
    path0 = os.path.dirname(__file__)
    path1 = os.path.join(path0, 'saveimage', file_outname)
    image2.save(path1)
    # here path1 is the path for MNIST_r to read the image, fileoutname is the␣
↪name of the image
    outputimage = (path1, file_outname)
    return outputimage
```

Through this function, we could change the image from base64 form to image with its size as 28*28px, and its form as binarized image. Therefore, now this image is available to the MNIST model as input.

Then the prediction process is similar as **code 1.3(In [29])**.

3.2.3 Cassandra and Return

The cassandra process is similar in 2.1, so we skip this process.

Let's start the return part. Since the shifting process is completed by ajax, here we have to return a json file(here the file is the output of prediction), which is readable for ajax, back to ajax **In [37],line [19]**. And as is written in **code 0.5,line[18,21]**, after receiving the 'success' status, ajax help turns to **'/result'**.

Here's the **'/result'** page setting.

[39]:
```
# 1.6 set answer page no.2
@app.route('/result', methods=['GET'])
def result():
    data = request.args.get('data')  # use GET to receive the data
    if not data:
        abort("error!")
    return render_template('output.html', target=data)
```

Since we use ajax to send data, here the request command is slightly different. But the output principle is similar.

To see the whole process on website page, please go to Result part.

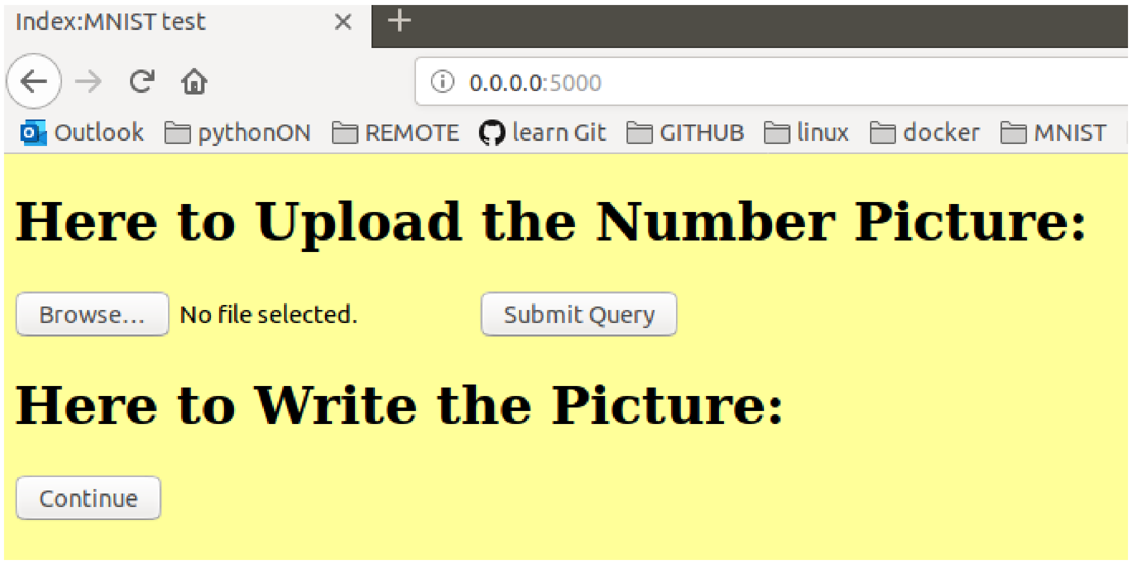
4 Result

The whole process on website is readable in my program as a video in **'/flaskandmnist/examplevideo/'[3]**. Here I would show the result in a series of pictures. Obviously, I would separate the result as two parts.

4.1 Result for file submitting

First, start the Cassandra container and the program container on the Terminal.**(if you submit the program onto docker. But if you want to run it on local host, you should run the flask_1.py as the engine)**

Click the url given. We will get to this page.


Index:MNIST test

0.0.0.0:5000

Outlook pythonON REMOTE learn Git GITHUB linux docker MNIST

Here to Upload the Number Picture:

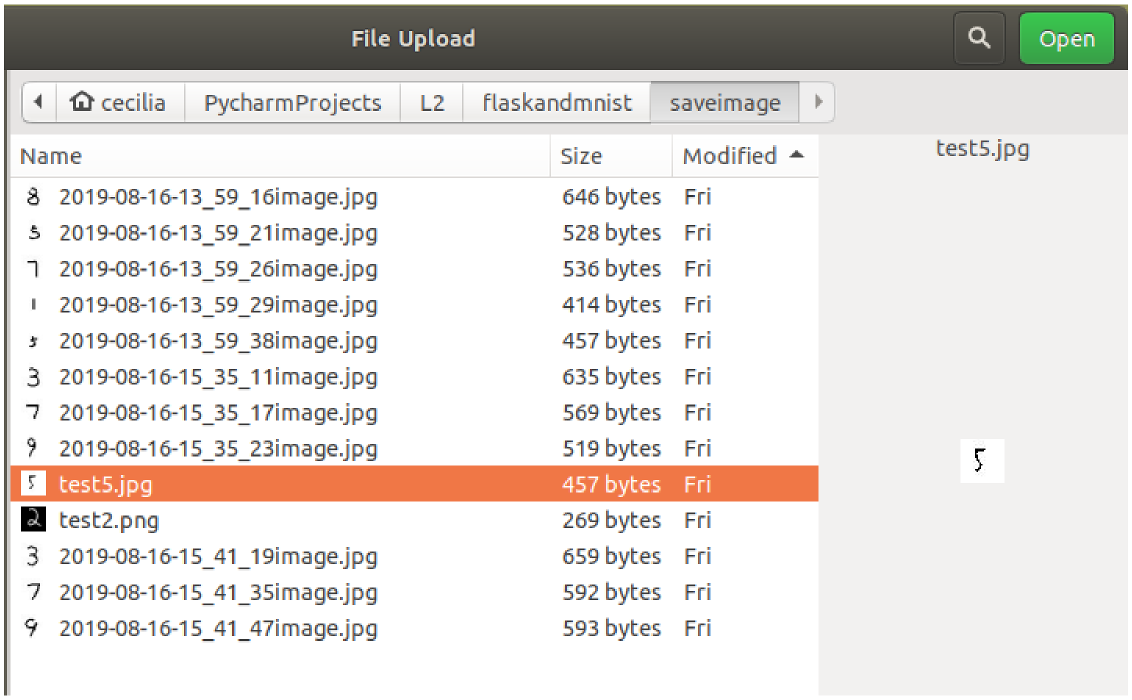
Browse... No file selected. Submit Query

Here to Write the Picture:

Continue

Here we test for the first function, so click ‘Brouse’ button and upload an image.

Attention: the image uploaded here fits the demand of MNIST model **(the example images are given in ‘/flaskandmnist/exampleimages/’)**. If you want to upload an image with normal type, you should set the image to ‘produceImage()’ funciton first. Here we submit a ‘5’ image and click ‘Submit Query’.

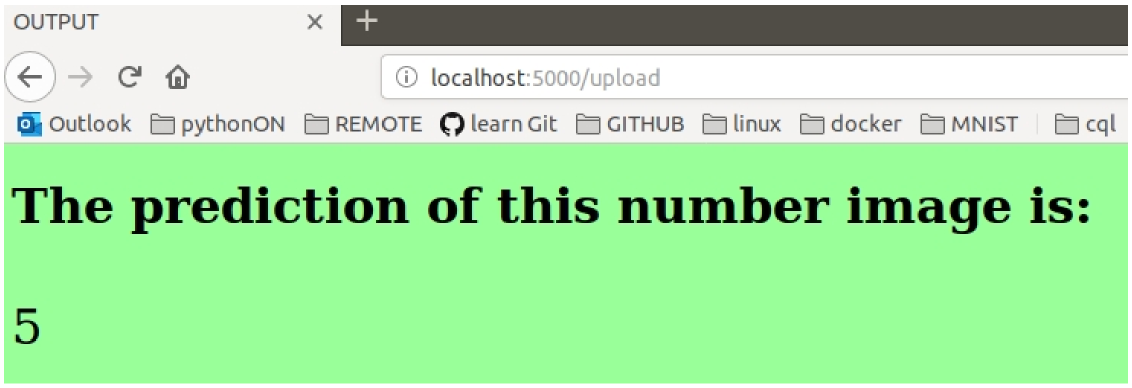

File Upload — Open

cecilia / PycharmProjects / L2 / flaskandmnist / saveimage

Name	Size	Modified
2019-08-16-13_59_16image.jpg	646 bytes	Fri
2019-08-16-13_59_21image.jpg	528 bytes	Fri
2019-08-16-13_59_26image.jpg	536 bytes	Fri
2019-08-16-13_59_29image.jpg	414 bytes	Fri
2019-08-16-13_59_38image.jpg	457 bytes	Fri
2019-08-16-15_35_11image.jpg	635 bytes	Fri
2019-08-16-15_35_17image.jpg	569 bytes	Fri
2019-08-16-15_35_23image.jpg	519 bytes	Fri
test5.jpg	457 bytes	Fri
test2.png	269 bytes	Fri
2019-08-16-15_41_19image.jpg	659 bytes	Fri
2019-08-16-15_41_35image.jpg	592 bytes	Fri
2019-08-16-15_41_47image.jpg	593 bytes	Fri

test5.jpg

Here comes the output:



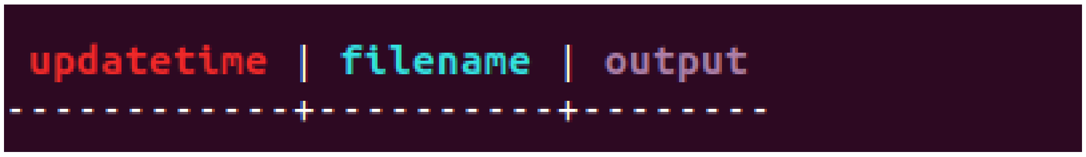
The prediction of this number image is:

5

Here I use cassandra2 as the name of the cassandra container, uploadinfo1 as the keyspace and info1 as the table name. To check if the info is really uploaded onto the Cassandra table, let's try these code on Terminal:

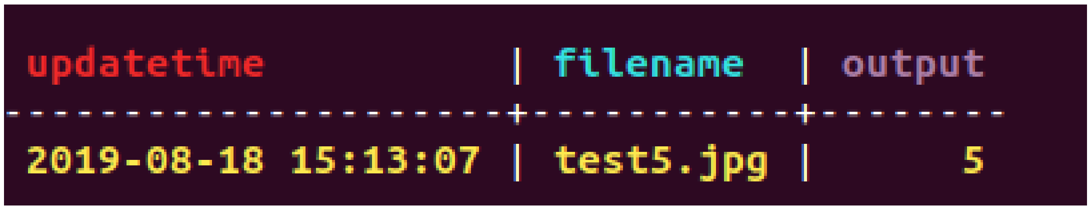
```
[44]: # ~$ docker exec -it cassandra2 cqlsh
      # cqlsh> use uploadinfo2;
      # cqlsh:uploadinfo2> select * from info1;
```

Before uploading, the table is like:



```
 updatetime | filename | output
------------+----------+--------
```

After uploading, the table becomes:



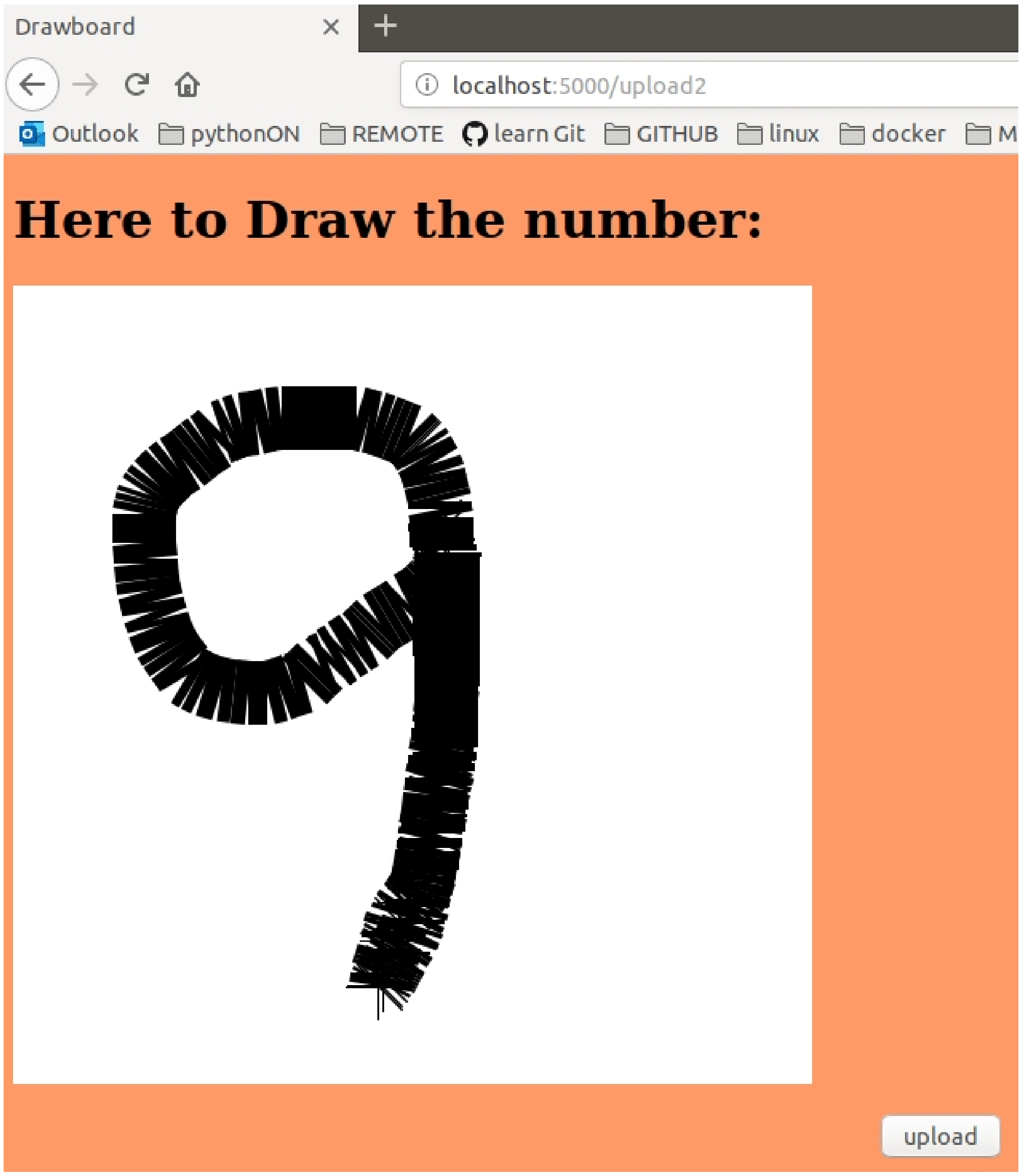
```
 updatetime          | filename  | output
---------------------+-----------+--------
 2019-08-18 15:13:07 | test5.jpg |      5
```

So it's a successful process.

4.2 Result for draw-board submitting

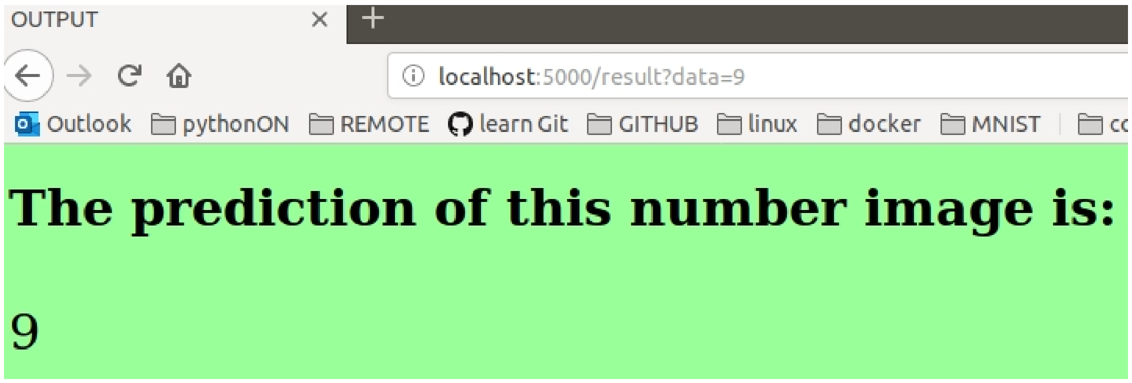
Then we test for the second function. Let's click 'Continue' first.

We come to a draw-board. Draw a '9' as example.



Here to Draw the number:

Click ‘upload’ to continue. And we get the result as:



The prediction of this number image is:

9

The Cassandra's output is similar to the result in function 1, so we skip this process.

So the process is also successful.

5 Concousion

As is said in the previous paragraphs, this program manages to use MNIST model from two different input, and it makes the process by operating on graphic website instead of codes. Personally, I make the most part of my effort on the realization of the canvas draw-board and the ajax interaction, since i've never learnt about the html front-end system before. So this program really helps a lot on learning new information and on having a deeper idea of the data-exchanging process. Also, the experience of using docker as a virtual container is entirely new to me, which enables me to get a stretch at the real operating system outside the campus. Thanks again to Dr.Zhang for the kind introduction.

Apart from these personal reflections, there're also points left to imporve for this project. On one hand, although the model's accuracy seems high enough, it would have problem figuring out the number sometimes. I think I could do further improvement on the picture transform process and on the model itself to get better accuracy. On the other, the canvas draw-board is still of basic mode, with unsmooth painting lines. It could also be managed during my further learning.

6 References and Links

[1]https://github.com/tensorflow/tensorflow/tree/r1.4/tensorflow/examples/tutorials/mnist

[2]http://yann.lecun.com/exdb/mnist/

[3]https://github.com/LetTheWorldGoos/Numberfigure.git

[4]https://blog.csdn.net/u013282737/article/details/85233408