ocr-api-gcp

Autor:

Kevin Cárdenas.

2023

Índice

1. Abstract

El código completo consta de varias partes que trabajan juntas para procesar imágenes de diferentes marcas y extraer información útil de ellas.

Comencemos con el archivo orchestrator.py. Este archivo contiene la función detect_text(), que es la función principal que se utiliza para procesar las imágenes y extraer la información relevante. Esta función utiliza la API de OCR (reconocimiento óptico de caracteres) de Google Cloud Vision para extraer texto de las imágenes.

La función detect_text() comienza leyendo la imagen de un bucket de almacenamiento en la nube de Google. A continuación, utiliza la API de OCR de Google Cloud Vision para extraer el texto de la imagen. Luego, basándose en el texto extraído, se invocan diferentes funciones para extraer información específica de diferentes marcas. Por último, se devuelve un diccionario con la información extraída.

Las funciones que se llaman dentro de detect_text() para extraer información específica de diferentes marcas son extraer_exito(), extraer_arturo_calle(), extraer_homecenter(), etc. Cada una de estas funciones utiliza expresiones regulares para buscar patrones específicos en el texto extraído de la imagen y extraer información relevante.

Además de la función detect_text(), el archivo orchestrator.py también contiene una serie de funciones auxiliares que se utilizan para la gestión de logs y la comunicación con RabbitMQ.

La función mensaje_trazabilidad es responsable de enviar un mensaje a RabbitMQ con información sobre la entrada y salida de la función detect_text. Esto se hace para tener un registro de las transacciones y para poder rastrear cualquier problema que pueda surgir. El mensaje contiene información como el nombre del archivo, la hora de procesamiento y los resultados de OCR.

La función enviar_mensaje es utilizada por la función mensaje_trazabilidad para enviar el mensaje a la cola de RabbitMQ.

La función enviar_a_crm es responsable de enviar información sobre el resultado del OCR a un sistema de CRM (Customer Relationship Management). Esto se hace para que los empleados puedan ver la información de manera más fácil y eficiente.

Por último, la función validacion_final se encarga de realizar una validación adicional en el resultado del OCR para asegurarse de que la información extraída es correcta. Esto se hace utilizando una serie de reglas específicas para cada marca. Si la validación es exitosa, se envía la información al sistema de CRM. Si no, se descarta.

El archivo main.py es el punto de entrada del programa. En este archivo se llama a la función connect_ssl() para establecer una conexión SSL con RabbitMQ y comenzar a recibir mensajes desde una cola específica. Cuando un mensaje se recibe en la cola, se llama a la función detect_text() para procesar la imagen contenida en el mensaje y extraer la información relevante. La información extraída se envía a otra cola específica para su posterior procesamiento.

2. Explicación del codigo

Dividiré el codigo en bloques e intentaré explicar la funcionalidad de cada uno.

2.1. LLamado a librerias y marcas parametrizadas

```
import os
from urllib import response
import pandas as pd
from re import compile, match
import requests
from os import listdir, getcwd
from os.path import isfile, join
import numpy as np
from datetime import datetime, timedelta
from google.cloud import vision
import io
import sys
sys.path.append('/home/despinosar/ocr_testing/marcas/')
sys.path.append('../marcas/')
from marcas.arturo_calle import extraer_arturo_calle
from marcas.bata import extraer_bata
from marcas.bosi import extraer_bosi
from marcas.calzatodo import extraer_calzatodo
from marcas.casaideas import extraer_casaideas
from marcas.crepes_y_waffles import extraer_crepes_y_waffles
from marcas.homecenter import extraer_homecenter
from marcas.decathlon import extraer_decathlon
from marcas.dollarcity import extraer_dollarcity
from marcas.el_corral import extraer_el_corral
from marcas.exito import extraer_exito
from marcas.frisby import extraer_frisby
from marcas.gef import extraer_gef
from marcas.happy_citty import extraer_happy_citty
from marcas.helados_popsy import extraer_helados_popsy
from marcas.juan_valdez import extraer_juan_valdez
from marcas.koaj import extraer_koaj
from marcas.medipiel import extraer_medipiel
from marcas.ishop import extraer_ishop
from marcas.HyM import extraer_HyM
from marcas.miniso import extraer_miniso
from marcas.naf_naf import extraer_naf_naf
from marcas.offcorss import extraer_offcorss
```

```
from marcas.patprimo import extraer_patprimo
from marcas.polo_club import extraer_polo_club
from marcas.rifle import extraer_rifle
from marcas.seven_seven import extraer_seven_seven
from marcas.studio_f import extraer_studio_f
from marcas.tennis import extraer_tennis
from marcas.totto import extraer_totto
from google.cloud import bigquery
import pika
import requests
import json
from google.oauth2 import service_account
import random
import ssl
import logging
```

En las primeras líneas del código, se importan diferentes paquetes de Python necesarios para el proyecto, como **os**, **pandas**, **numpy**, **datetime**, **google.cloud.vision**, **sys**, **requests**, **json**, **google.oauth2.service_account** y **logging**.

- **os**: Esta librería proporciona una forma portátil de utilizar la funcionalidad del sistema operativo subyacente, como leer o escribir en el sistema de archivos, manipular rutas, trabajar con variables de entorno, etc.
- **pandas**: Esta librería se utiliza para el análisis de datos y la manipulación de los mismos. Es especialmente útil para trabajar con datos tabulares y para realizar operaciones de limpieza y transformación de los datos.
- **numpy**: Esta librería se utiliza para realizar operaciones matemáticas y numéricas. Proporciona una estructura de datos llamada **array** que permite realizar operaciones vectoriales y matriciales de manera eficiente.
- **datetime**: Esta librería proporciona clases para trabajar con fechas y tiempos.
- **google.cloud.vision**: Esta librería proporciona una interfaz para utilizar el servicio de reconocimiento de imágenes de Google Cloud Vision.
- **sys**: Esta librería proporciona acceso a algunas variables y funciones utilizadas o mantenidas por el intérprete de Python.
- **requests**: Esta librería se utiliza para realizar solicitudes HTTP.
- **json**: Esta librería se utiliza para trabajar con objetos JSON.
- **google.oauth2.service_account**: Esta librería proporciona una forma de autenticarse con una cuenta de servicio de Google Cloud Platform.
- **logging**: Esta librería se utiliza para registrar mensajes de depuración, información, advertencia, error, etc. en una aplicación. Es útil para realizar un seguimiento de lo que está sucediendo en una aplicación y para depurar problemas.

A continuación, se importan diferentes módulos definidos en el directorio `marcas`, que contienen funciones específicas para extraer información que buscamos de facturas de cada marca. Cada función es específica para cada marca y utiliza técnicas de web scraping para extraer la información buscada de cada marca de la marca.

2.2. Creación de instancias

```
client = vision.ImageAnnotatorClient()
bqclient = bigquery.Client()
dataset_id, table_id = str(os.environ["bigquery_dataset"]), str(os.environ["bigquery_logs"])
table_ref = bqclient.dataset(dataset_id).table(table_id)
table = bqclient.get_table(table_ref)

dataset_id, table_id = str(os.environ["bigquery_dataset"]), str(os.environ["bigquery_facturas_sin_pc"])
table_ref_facturas = bqclient.dataset(dataset_id).table(table_id)
facturas_sin_puntos = bqclient.get_table(table_ref_facturas)

table_id =  str(os.environ["bigquery_estado_marcas"])
table_ref_estado_marcas = bqclient.dataset(dataset_id).table(table_id)
facturas_sin_estado_marcas = bqclient.get_table(table_ref_estado_marcas)
```

El código crea una instancia del cliente de Google Cloud Vision y un cliente de BigQuery. Luego, define los valores de dataset_id y table_id a partir de variables de entorno y utiliza estos valores para crear una referencia a una tabla en BigQuery. A continuación, se utiliza la referencia de tabla para obtener la tabla en sí misma. Luego, se repite este proceso para dos tablas adicionales en BigQuery.

2.3. Validación final

```
def validacion_final(resultados, mensaje, telefono):
    intencion = ''
    if resultados['marca'] != '': #Encontro la marca?
        if ((resultados["mall"] == '') or (resultados["numero_factura"] == '') or (telefono == '') or
         (resultados["register_date"] == '') or (resultados["session_id"] == '') or (resultados["total"] == '')): #La deteccion
     tiene nulos
            mensaje = 'Faltan datos obligatorios'
            intencion = 'OCR_Operador'
        else:
            if (pd.to_datetime(resultados['register_date']) < (datetime.now() - timedelta(days = 31))) or (pd.to_datetime(
     resultados['register_date']) > datetime.now()):
                mensaje = 'No se pueden registrar facturas con mas de 1 mes de antiguedad'
                intencion = 'Error_Validacion'
    else:
        mensaje = 'Marca no encontrada'
```

```
        intencion = 'OCR_Operador'
    return resultados, mensaje, intencion
```

Esta función llamada validacion_final recibe tres parámetros: resultados, mensaje, telefono.

Primero se verifica si se encontró una marca en los resultados obtenidos. Si no se encontró una marca, se establece un mensaje de error y una intención asociada al error.

Si se encontró una marca, se verifican que no haya valores nulos para los campos mall, numero_factura, register_date, session_id, total y telefono. Si algún valor es nulo, se establece un mensaje de error y una intención asociada al error.

Si no hay valores nulos, se verifica si la fecha de registro de la factura es mayor a un mes de antigüedad o menor a la fecha actual. Si la fecha no está dentro del rango permitido, se establece un mensaje de error y una intención asociada al error.

En caso contrario, se devuelve resultados, el mensaje actualizado y una cadena vacía como intención.

2.4. Enviar a crm

```
def enviar_a_crm(resultados, mensaje, channel):
    registra = False #Por defecto no se ha registrado la factura
    if ((mensaje == 'Faltan datos obligatorios') or (mensaje == 'Marca no encontrada') or (mensaje == 'No se pueden registrar
     facturas con mas de 1 mes de antiguedad')):
        code = 201
        api_crm_response = {'code':201, 'Message':mensaje}
        pass
    else:
        session = requests.Session()
        session.auth = (str(os.environ["crm_user"]), str(os.environ["crm_pw"]))
        resultados['register_date'] = str(resultados['register_date'])
        api_crm_response = session.post(str(os.environ["crm_url"]), data = resultados).json()
        registra = api_crm_response['Result']
        code = api_crm_response['Code']
        if (registra == True) and (code == 18): #Registro exitoso
            mensaje = api_crm_response['Message']
            try: #Verificar si los puntos llegaron
                int(api_crm_response['Puntos'].split()[-3])
            except: #En caso de que no lleguen mandar a la tabla de envio manual
                api_crm_response['Message'] = "SIN PUNTOS " + mensaje
                code = str(code) + "-2"
                mensaje = api_crm_response['Message']
                rows_to_insert = [(json.dumps(resultados, ensure_ascii=False), str(datetime.now().date()))]
                errors = bqclient.insert_rows(facturas_sin_puntos, rows_to_insert)
                #Trazabilidad CRM
                if errors != []: #Si hay errores al cargar datos, mandarlos tambien a BigQuery pero con ese detalle.
                    rows_to_insert = [(resultados['mall'], resultados['marca'], "200", "Error al registrar facturas sin puntos", ""
```

```
    , str(datetime.now().date()), "False")]
                errors = bqclient.insert_rows(table, rows_to_insert)
        try:
            mensaje_trazabilidad(channel = channel, tipo = 'OUT', applicationId = 'CRM-VIVA', data = resultados, status = 'OK', trace = 'Registro exitoso',
                                 messagesIn = "0", messagesOut = "1", messagesError = "0", mensaje_resultado = api_crm_response)
        except:
            pass
    else:
        mensaje = "Error registro CRM - " + api_crm_response['Message']
        try:
            mensaje_trazabilidad(channel = channel, tipo = 'OUT', applicationId = 'CRM-VIVA', data = resultados, status = 'ERROR', trace = 'Registro no exitoso',
                                 messagesIn = "0", messagesOut = "0", messagesError = "1", mensaje_resultado = api_crm_response)
        except:
            pass
        print(mensaje)
    return mensaje, registra, code, api_crm_response
```

El código define una función llamada enviar_a_crm que recibe tres parámetros: resultados, mensaje y channel.

Dentro de la función se inicializa una variable booleana llamada registra en False. Luego se realiza una serie de validaciones para verificar si es posible enviar la información del OCR al CRM.

Si el mensaje recibido es Faltan datos obligatorios, Marca no encontrada o No se pueden registrar facturas con más de 1 mes de antiguedad, la función retorna un mensaje de error y no continúa con el proceso.

En caso contrario, se realiza una petición POST al CRM utilizando una sesión autenticada y se envía la información contenida en resultados. Si el registro es exitoso, se verifica si se recibieron los puntos correspondientes y si es así, se retorna un mensaje de éxito. En caso contrario, se guarda la información en una tabla de BigQuery para su posterior revisión manual.

Finalmente, se envía un mensaje de trazabilidad a través de un servicio de mensajería para registrar el resultado del registro en el CRM. La función retorna un mensaje de éxito o error, el valor de registra, el código de respuesta de la API y la respuesta de la API en formato JSON.

2.5. Enviar mensaje

```
def enviar_mensaje(registra, telefono, api_crm_response, intencion, code, channel, mensaje):
    url = str(os.environ["botmaker_url"])
    headers = {
            'access-token': str(os.environ["botmaker_token"]),
            'Accept': 'application/json',
            'Content-Type': 'application/json'
            }
    if registra == True:
```

```
        eventos = np.arange(len(api_crm_response['Data']))
        mensaje_eventos = ""
        for i in eventos:
            frase = " \n - {boletas} boletas para {evento}".format(
                                boletas = api_crm_response['Data'][i]['BALLOT'],
                                evento = api_crm_response['Data'][i]['EVENT_NAME'].capitalize()
                                                        )
            mensaje_eventos = mensaje_eventos + frase
        mensaje_eventos = mensaje_eventos.strip()
        try: #Verificar si viene con Puntos Colombia
            puntos = api_crm_response['Puntos'].split()[-3]
            int(puntos) #Comprobar que sea numerico
        except: #En caso de que no venga, mandar 0 Puntos Colombia,
            puntos = 0
        payload = json.dumps({
        "chatPlatform": "whatsapp",
        "chatChannelNumber": api_crm_response['chatChannelNumber'],
        "platformContactId": telefono,
        "ruleNameOrId": "Intent",
        "params": {
            "N_Puntos_Colombia": puntos,
            "N_Eventos": mensaje_eventos
        }})
        response = requests.request("POST", url, headers=headers, data=payload)
        if response.json()['problems'] == None:
            try:
                mensaje_trazabilidad(channel = channel, tipo = 'OUT', applicationId = 'BotMaker', data = payload, status = 'OK',
    trace = 'Notificacion exitosa',
                                messagesIn = "0", messagesOut = "1", messagesError = "0", mensaje_resultado = response.json())
            except:
                pass
        else:
            try:
                mensaje_trazabilidad(channel = channel, tipo = 'OUT', applicationId = 'BotMaker', data = payload, status = 'ERROR',
     trace = 'Notificacion no exitosa',
                                messagesIn = "0", messagesOut = "0", messagesError = "1", mensaje_resultado = response.json())
            except:
                pass
    else: #Casos de error, basados en la intencion
        if code == 13: #Factura registrada previamente
            payload = json.dumps({
                "chatPlatform": "whatsapp",
                "chatChannelNumber": api_crm_response['chatChannelNumber'],
```

```
                "platformContactId": telefono,
                "ruleNameOrId": "Error_Validacion",
                "params": {"MsgOCR":'La factura ya se encuentra registrada', "CodeOCR":0}
                })
            response = requests.request("POST", url, headers=headers, data=payload)
        if intencion == 'Error_Validacion': #Errores de registro en CRM
            payload = json.dumps({
            "chatPlatform": "whatsapp",
            "chatChannelNumber": api_crm_response['chatChannelNumber'],
            "platformContactId": telefono,
            "ruleNameOrId": "Error_Validacion",
            "params": {"MsgOCR": api_crm_response['Message'], "CodeOCR":0}
            })
            response = requests.request("POST", url, headers=headers, data=payload)
        elif intencion == 'OCR_Operador': #No se lee factura adecuadamente
            payload = json.dumps({
            "chatPlatform": "whatsapp",
            "chatChannelNumber": api_crm_response['chatChannelNumber'],
            "platformContactId": telefono,
            "ruleNameOrId": "OCR_Operador"
            })
            response = requests.request("POST", url, headers=headers, data=payload)
        else: #No mande nada
            return
    return response.json()
```

La función enviar_mensaje toma como entrada los siguientes parámetros: registra, telefono, api_crm_response, intencion, code, channel, mensaje. Esta función se encarga de enviar un mensaje al usuario a través de la plataforma de WhatsApp mediante la integración con la API de BotMaker.

El código comienza definiendo la variable url con la URL de la API de BotMaker y las variables headers que contiene la información necesaria para realizar una solicitud HTTP.

A continuación, se verifica si el parámetro registra es verdadero o falso. Si es verdadero, se recorre la lista api_crm_response['Data'] para obtener la información de los eventos y boletas registrados, construyendo una cadena de texto con esta información en la variable mensaje_eventos. Luego, se verifica si el campo 'Puntos' está presente en api_crm_response. Si es así, se extrae el número de puntos de la cadena y se asigna a la variable puntos. En caso contrario, se asigna el valor 0 a puntos.

Luego se construye un objeto JSON payload con la información del mensaje a enviar. Este objeto contiene la información del canal de chat, el número de contacto, la intención, el número de puntos y la lista de eventos registrados. Este objeto se envía como carga útil en una solicitud HTTP POST a la API de BotMaker.

Si la respuesta de la API de BotMaker no contiene problemas, se llama a la función mensaje_trazabilidad para registrar el mensaje enviado en la base de datos de trazabilidad. En caso contrario, también se llama a mensaje_trazabilidad, pero con un

estado de error.

Si el parámetro registra es falso, se verifica la intención o el código de error para construir un mensaje de respuesta específico. Si el código de error es 13, se construye un mensaje que indica que la factura ya está registrada. Si la intención es 'Error_Validacion', se construye un mensaje con la información del error devuelto por la API de CRM. Si la intención es 'OCR_Operador', se construye un mensaje indicando que la factura no se pudo leer adecuadamente. En caso contrario, no se envía ningún mensaje.

En todos los casos, la función devuelve el resultado de la solicitud HTTP POST como un objeto JSON.

2.6. Mensaje de trazabilidad

```
def mensaje_trazabilidad(channel, tipo, applicationId, data, status, trace,
    messagesIn, messagesOut, messagesError, mensaje_resultado):

    transactionId = str(datetime.now().date()) + '-' + str(random.randint(1,100000))
    fecha_hora = str(datetime.now()).split('.')[0] #Tomar solo fecha
    payload = {
    "transactionId": transactionId,
    "integrationName": "ocr-api-gcp",
    "domainName": "ocr-viva",
    "operation": "processing",
    "type": tipo,
    "timeStamp": fecha_hora,
    "event":
    {
    "header": {
    "transactionId": transactionId,
    "applicationId": applicationId,
    "transactionDate": fecha_hora,
    "flexField":[]
    },
    "data": data
    },
    "status": status,
    "trace": trace,
    "messagesIn": messagesIn,
    "messagesBlocks": "1",
    "messagesOut": messagesOut,
    "messagesError": messagesError,
    "messagesFilter": "0",
    "response_time": "0",
    "mensaje_resultado": mensaje_resultado
```

```
    }
    channel.basic_publish(exchange=str(os.environ["rabbit.traceabilityaks.exchange"]), routing_key=str(os.environ["rabbit.traceabilityaks.routingkey"]), body=json.dumps(payload, indent = 4))
    return True
```

La función mensaje_trazabilidad se encarga de enviar un mensaje de trazabilidad para el registro y monitoreo de las transacciones que ocurren en la aplicación.

Toma como parámetros channel, tipo, applicationId, data, status, trace, messagesIn, messagesOut, messagesError y mensaje_resultado.

Crea un ID de transacción y una marca de tiempo, y luego construye un diccionario payload que contiene información relevante sobre la transacción y su estado. Finalmente, publica el mensaje de trazabilidad en un exchange de RabbitMQ a través de un canal básico usando los valores de enrutamiento y exchange que se obtienen de variables de entorno.

Devuelve True al final de la función.

2.7. Detectar texto

```
def detect_text(ch, method, properties, path):
    #Conexion a cola de logs
    logging.basicConfig(level=logging.INFO)
    context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
    ssl_options = pika.SSLOptions(context, str(os.environ["rabbit.traceabilityaks.host.cloud"]))
    credentials = pika.PlainCredentials(str(os.environ["rabbit.traceabilityaks.user"]), str(os.environ["rabbit.traceabilityaks.password"]))
    conn_params = pika.ConnectionParameters(host=str(os.environ["rabbit.traceabilityaks.host.cloud"]),port=str(os.environ["rabbit.traceabilityaks.port"]), client_properties={'connection_name': 'Conexion_OCR',
    }, ssl_options=ssl_options, virtual_host=str(os.environ["rabbit.traceabilityaks.virtualhost"]), credentials = credentials)
    connection = pika.BlockingConnection(conn_params)
    channel = connection.channel() #Conexion realizada
    #Extraer informacion del json
    try:
        path = json.loads(path.decode("utf-8").strip("b"))
        telefono = path['clientPhoneNumber'] #Telefono del cliente
        sessionId = path['sessionId'] #SessionId conversacion
        cedula = path['clientDocument']
        #Mensaje trazabilidad IN
        try:
            mensaje_trazabilidad(channel = channel, tipo = 'IN', applicationId = 'AKS-RabbitMQ', data = path, status = 'OK', trace = 'Lectura exitosa',
                            messagesIn = "1", messagesOut = "0", messagesError = "0", mensaje_resultado = path)
        except:
            pass
```

```
    except:
        try:
            mensaje_trazabilidad(channel = channel, tipo = 'IN', applicationId = 'AKS-RabbitMQ', data = path, status = 'ERROR',
 trace = 'Lectura no exitosa',
                            messagesIn = "0", messagesOut = "0", messagesError = "1", mensaje_resultado = path)
        except:
            pass
    try: #Tratar de detectar la imagen en GCS
        text = 'Null'
        print("leyendo imagen")
        image = vision.Image(source = vision.ImageSource(image_uri = path['url'])) #Lectura API Vision
        response = client.document_text_detection(image=image) #Deteccion de texto
        texts = response.text_annotations #Extraccion del texto plano
        text = texts[0].description.upper() #Texto en mayusculas
    except Exception as E:
        print("Error de lectura", E)
    if text != 'Null': #Leer e interpretar la imagen
        #Definir json inicial
        print("interpretando")
        resultados = {
            "canal_compra": "",
            "canal_registro": "",
            "client_document": "",
            "mall": "",
            "marca": "",
            "medio_pago": "",
            "numero_factura": "",
            "register_date": "",
            "session_id": "",
            "text": "",
            "total": ""}
        #Consulta al estado de las marcas:
        filas = bqclient.list_rows(
        facturas_sin_estado_marcas,
        selected_fields = [
            bigquery.SchemaField('Marca', 'STRING'),
            bigquery.SchemaField('Estado', 'STRING')
            ],
        )
        estado_marcas = filas.to_dataframe()
        if (("ALMACENES EXITO" in text) or ("8909006089" in text)):
            if (estado_marcas[estado_marcas['Marca'] == 'ALMACENES EXITO']['Estado'].values[0]=='ON'):
                resultados = extraer_exito(text, bqclient=bqclient)
```

```
    elif (("ARTURO CALLE" in text) or ("900.342.297-2" in text)):
        if (estado_marcas[estado_marcas['Marca'] == 'ARTURO CALLE']['Estado'].values[0]=='ON'):
            resultados = extraer_arturo_calle(text)
    elif (("BATA" in text) and ("890801339-8" in text)):
        if (estado_marcas[estado_marcas['Marca'] == 'BATA']['Estado'].values[0]=='ON'):
            resultados = extraer_bata(text)
    elif (("BOSI" in text) or ("800.165.720-5" in text)):
        if(estado_marcas[estado_marcas['Marca'] == 'BOSI']['Estado'].values[0]=='ON'):
            resultados = extraer_bosi(text)
    elif (("CALZATODO" in text) or ("805.004.875-6" in text)):
        if(estado_marcas[estado_marcas['Marca'] == 'CALZATODO']['Estado'].values[0]=='ON'):
            resultados = extraer_calzatodo(text)
    elif (("DECATHLON" in text) or ("900868271-1" in text)):
        if (estado_marcas[estado_marcas['Marca'] == 'DECATHLON']['Estado'].values[0]=='ON'):
            resultados = extraer_decathlon(text)
    elif (("DOLLARCITY" in text) or ("9009432434" in text)):
        if (estado_marcas[estado_marcas['Marca'] == 'DOLLARCITY']['Estado'].values[0]=='ON'):
            resultados = extraer_dollarcity(text)
    elif (("EL CORRAL" in text) or ("860533413-6" in text)):
        if (estado_marcas[estado_marcas['Marca'] == 'EL CORRAL']['Estado'].values[0]=='ON'):
            resultados = extraer_el_corral(text)
    elif (("FRISBY" in text) or ("891408584" in text)):
        if (estado_marcas[estado_marcas['Marca'] == 'FRISBY']['Estado'].values[0]=='ON'):
            resultados = extraer_frisby(text)
    elif (("GEF" in text) or ("890901672-5" in text)):
        if (estado_marcas[estado_marcas['Marca'] == 'GEF']['Estado'].values[0]=='ON'):
            resultados = extraer_gef(text)
    elif (("HAPPY CITTY" in text) or ("890.930.448-5" in text)):
        if (estado_marcas[estado_marcas['Marca'] == 'HAPPY CITTY']['Estado'].values[0]=='ON'):
            resultados = extraer_happy_citty(text)
    elif (("HELADOS POPSY" in text) or ("860.053.831-1" in text)):
        if (estado_marcas[estado_marcas['Marca'] == 'HELADOS POPSY']['Estado'].values[0]=='ON'):
            resultados = extraer_helados_popsy(text)
    elif (("JUAN VALDEZ" in text) or ("830.112.317-1" in text)):
        if (estado_marcas[estado_marcas['Marca'] == 'JUAN VALDEZ']['Estado'].values[0]=='ON'):
            resultados = extraer_juan_valdez(text)
    elif (("KOAJ" in text) or ("901407289-8" in text)):
        if (estado_marcas[estado_marcas['Marca'] == 'KOAJ']['Estado'].values[0]=='ON'):
            resultados = extraer_koaj(text)
    elif (("MEDIPIEL" in text) or ("811.041.214-7" in text)):
        if (estado_marcas[estado_marcas['Marca'] == 'MEDIPIEL']['Estado'].values[0]=='ON'):
            resultados = extraer_medipiel(text)
    elif (("H&M" in text) or ("900924527" in text)):
```

```
            if (estado_marcas[estado_marcas['Marca'] == 'H&M']['Estado'].values[0]=='ON'):
                resultados = extraer_HyM(text)
        elif (("CASAIDEAS" in text) or ("900395158" in text)):
            if (estado_marcas[estado_marcas['Marca'] == 'CASAIDEAS']['Estado'].values[0]=='ON'):
                resultados = extraer_casaideas(text)
        elif (("CREPES Y WAFFLES" in text) or ("800180330" in text)):
            if (estado_marcas[estado_marcas['Marca'] == 'CREPES Y WAFFLES']['Estado'].values[0]=='ON'):
                resultados = extraer_crepes_y_waffles(text)
        elif (("HOMECENTER" in text) or ("800242106" in text)):
            if (estado_marcas[estado_marcas['Marca'] == 'HOMECENTER']['Estado'].values[0]=='ON'):
                resultados = extraer_homecenter(text)
        elif (("ISHOP" in text) or ("900277370" in text)):
            if (estado_marcas[estado_marcas['Marca'] == 'ISHOP']['Estado'].values[0]=='ON'):
                resultados = extraer_ishop(text)
        elif (('MINISO' in text) or ("901.137.699-5" in text)):
            if (estado_marcas[estado_marcas['Marca'] == 'MINISO']['Estado'].values[0]=='ON'):
                resultados = extraer_miniso(text)
        elif (('NAF NAF' in text) or ("811014191" in text)):
            if (estado_marcas[estado_marcas['Marca'] == 'NAF NAF']['Estado'].values[0]=='ON'):
                resultados = extraer_naf_naf(text)
        elif (('OFFCORSS' in text) or ("802002267" in text)):
            if (estado_marcas[estado_marcas['Marca'] == 'OFFCORSS']['Estado'].values[0]=='ON'):
                resultados = extraer_offcorss(text)
        elif (('PATPRIMO' in text) and ("860503159-1" in text)):
            if (estado_marcas[estado_marcas['Marca'] == 'PATPRIMO']['Estado'].values[0]=='ON'):
                resultados = extraer_patprimo(text)
        elif (('POLO CLUB' in text) or ("900364648-9" in text)):
            if (estado_marcas[estado_marcas['Marca'] == 'POLO CLUB']['Estado'].values[0]=='ON'):
                resultados = extraer_polo_club(text)
        elif (('RIFLE' in text) or ("860353709" in text)):
            if (estado_marcas[estado_marcas['Marca'] == 'RIFLE']['Estado'].values[0]=='ON'):
                resultados = extraer_rifle(text)
        elif (('SEVEN/SEVEN' in text) and ("860503159-1" in text)):
            if (estado_marcas[estado_marcas['Marca'] == 'SEVEN SEVEN']['Estado'].values[0]=='ON'):
                resultados = extraer_seven_seven(text)
        elif (('STF GROUP' in text) or ("805003626-4" in text)):
            if (estado_marcas[estado_marcas['Marca'] == 'STUDIO F']['Estado'].values[0]=='ON'):
                resultados = extraer_studio_f(text)
        elif (('TOTTO' in text) or ("800.020.706" in text)):
            if (estado_marcas[estado_marcas['Marca'] == 'TOTTO']['Estado'].values[0]=='ON'):
                resultados = extraer_totto(text)
        elif (('TENNIS' in text) or ("890920043-3" in text)):
            if (estado_marcas[estado_marcas['Marca'] == 'TENNIS']['Estado'].values[0]=='ON'):
```

```
151                 resultados = extraer_tennis(text)
152         #Registro de logs
153         resultados['client_document'] = cedula
154         resultados['session_id'] = sessionId
155         print('validacion final: ', resultados) #comentario
156         resultados, mensaje, intencion = validacion_final(resultados, mensaje = '', telefono = telefono) #Validar datos
157         print('resultados: ', resultados, mensaje, intencion, '\n')
158         mensaje, registra, code, api_crm_response = enviar_a_crm(resultados, mensaje, channel) #CRM
159         print('salida de la funcion: ', mensaje, registra, code, api_crm_response ) #comentario
160         api_crm_response['chatChannelNumber'] = path['chatChannelNumber'] #A~nadir telefono del mall al json de respuesta
161         response = enviar_mensaje(registra, telefono, api_crm_response, intencion, code, mensaje, channel)
162         rows_to_insert = [(resultados['mall'], resultados['marca'], code, mensaje, resultados['session_id'], str(datetime.now().
    date()), str(registra))]
163     else: #En caso de que no aparezca la imagen registrarlo en logs de BigQuery
164         rows_to_insert = [("", "", "202", "OCR no lee factura", "", str(datetime.now().date()), "False")]
165         mensaje = 'Fallo en el registro'
166     errors = bqclient.insert_rows(table, rows_to_insert)
167     if errors != []: #Si hay errores al cargar datos, mandarlos tambien a BigQuery pero con ese detalle.
168         rows_to_insert = [("", "", "0", "Error al registrar log", "", str(datetime.now().date()), "False")]
169         errors = bqclient.insert_rows(table, rows_to_insert)
170     return print(mensaje)
```

2.8. Explicación del main

El código principal comienza importando la función "detect_text"del módulo .ºrchestrator". Luego, se define una función çonnect_ssl"que establece una conexión segura con RabbitMQ utilizando las credenciales y opciones SSL especificadas en las variables de entorno. A continuación, se establece una cola de RabbitMQ y se especifica la función "detect_textçomo la función de devolución de llamada que se ejecutará cuando se reciba un mensaje en la cola. Finalmente, se inicia el consumo de la cola y se espera a que lleguen los mensajes.

En el bloque ïf name == 'main':"se llama a la función çonnect_ssl", lo que significa que el código principal de este programa es establecer una conexión con RabbitMQ y escuchar mensajes en una cola. Cuando se recibe un mensaje, se llama a la función "detect_text"para procesar la imagen y extraer la información relevante de ella.