Politecnico di Torino

Politecnico di Torino
Corso di Laurea in
Ingegneria Informatica
A.A 2020/2021

Francesco Guarneri
S248967

Monografia di tirocinio

Sommario

Introduzione

Il tirocinio è stato svolto presso la AEC Soluzioni S.R.L., la quale si occupa di digitalizzazione dei processi produttivi industriali; è difatti una startup rivolta all'industria 4.0.

Il mio ruolo in questa azienda è stato il Data Analyst, nello specifico ho trattato l'analisi di flussi di dati ad alta frequenza utilizzando diversi linguaggi di programmazione e diversi tipi di database. Frequente anche l'uso di hardware esterno, nella fattispecie: un Raspberry Pi 4 e un nVidia Jetson Nano.

Non vi è stato un progetto singolo, ma tutti hanno come substrato l'analisi dei dati, tranne il primo che riguarda l'OCR e l'image preprocessing (una sorta di progetto di warming up). In questa fase introduttiva riporterò solo la lista di essi con un breve accenno descrittivo. Successivamente verranno descritti singolarmente in maniera dettagliata.

- ***PyOCRBusinessCard***, una webapp scritta in Python con lo scopo di convertire immagini di badge aziendali in file JSON contenenti svariate informazioni. Librerie utilizzate: OpenCV (elaborazione immagini), PyTesseract (conversione in testo).

- ***RedBot***, un'app scritta usando il runtime Node.JS e Redis (un in-memory data structure store), quest'ultimo utilizzato come database. Svolge la funzione di bridge in quanto inoltra (sottoscrivendosi) le pubblicazioni di Redis ad un bot Telegram. Il concept era quello di ricevere sul bot, in formato stringa, i JSON nel DB provenienti da PyOCRBusinessCard, il progetto sopra descritto. In realtà è però adattabile a qualunque tipo di dato che possa essere riportato in forma testuale (stringa, lista, dizionari, etc..).

- ***GreenLens***, una webapp che sfrutta il protocollo MQTT per mostrare i dati ricevuti da un sensore per piante. É inoltre presente in parallelo uno script Python che pusha gli stessi dati (si sottoscrive allo stesso publisher cui si sottoscrive la webapp) verso un database InFlux (di tipo time series).

- ***FluxBridge***. Continuando lo studio di InFlux, questa webapp scritta in Python utilizzando Flask per il server side ha il compito di gestire delle richieste HTTP POST provenienti dal database per poi inoltrarle verso un bot Telegram.

- ***JpianoPush***. Questo programma, cui è presente sia una versione in Python che in Java, legge da un logfile dell'azienda in continua scrittura (dunque dinamicamente), converte le informazioni utili in JSON e pusha il tutto verso un cloud bridge dell'azienda. Queste informazioni vengono poi inoltrate verso InFlux.
  Ho inoltre simulato la lettura dinamica offline. Per tale scopo ho utilizzato parallellamente un altro script Python che genera randomicamente le stringhe del logfile; i dati prodotti seguono una distribuzione gaussiana di cui è possibile settare i valori.

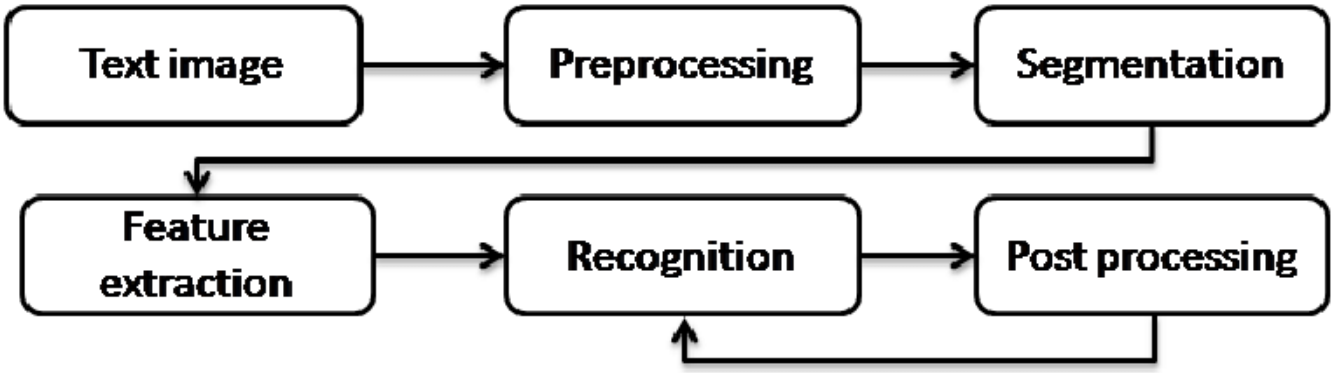
PyOCRBusinessCard
Chapter 0

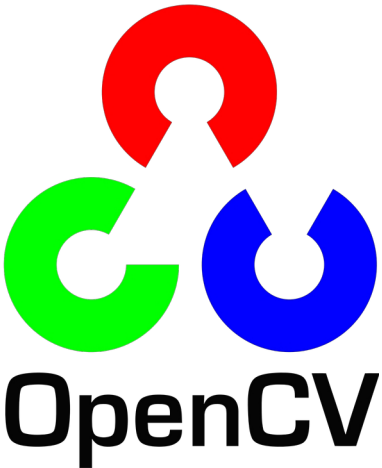
Concept
Optical Character Recognition (OCR) technology

Il principale obiettivo di un sistema OCR è quello di riconoscere il testo da una immagine e di convertirlo in formato ASCII oppure Unicode. Per avvalersi di ciò, un OCR deve seguire una fase di training per il riconoscimento delle lettere.

Coloured image →Gray Scale image →Binary image

Il meccanismo è il seguente: vengono forniti dei sample delle immagini corredate dall'equivalente carattere ASCII/Unicode in modo che il sistema le possa confrontare con quelle che avrà in input.
Il tasso di riconoscimento dell'alfabeto latino è ormai superiore al 99%, pertanto essi si rivelano strumenti estremamente efficaci.





Come però lo schema ben evidenzia, l'immagine deve prima seguire una fase di preprocessing, al fine di facilitare il riconoscimento da parte dell'OCR.

In ***PyOCRBusinessCard*** vanno dunque a congiungersi due librerie per la realizzazione di questo sistema: ***OpenCV***, che si occupa del preprocessing delle immagini, e ***PyTesseract***, che si occupa delle fasi successive presenti nello schema sopra fino alla recognition. Il post processing è stato invece implementato da me tramite regex e metodi più "from scratch" per filtrare le informazioni, in quanto non tutte quelle presenti nei badge andranno a contribuire nella costruzione del JSON.

Difatti, come riportato nello script Python, il dizionario che verrà dumpato in .json contiene i seguenti campi:

```
card = {"Type": "itemnote_create",
        "Context": "OP",
        "note": "", "description": "",
        "type": "visitor",
        "station":""}
```

Inizialmente l'immagine viene letta tramite una funzione presente nella libreria di OpenCV:

```
img = cv.imread(frameName)
```

Successivamente, utilizzando NumPy (una libreria Python con funzioni matematiche di alto livello e possibilità di utilizzare matrici multidimensionali), l'immagine viene importata in un numpy array le cui celle conterranno il valore di ogni pixel.

NB: da ora in avanti con le parole 'immagine' o 'foto' ci si riferirà sempre al numpy array generato.

```
img = np.array(img,dtype=np.uint8)
```

(Il data type dell’array in questo caso è 8-bit unsigned integer)

52	**55**	**61**	**59**	**79**	**61**	**76**	**61**
62	59	55	104	94	85	59	71
63	65	66	113	144	104	63	72
64	70	70	126	154	109	71	69
67	73	68	106	122	88	68	68
68	79	60	70	77	66	58	75
69	85	64	58	55	61	65	83
70	87	69	68	65	73	78	90

8-bit grayscale image, 8x8 matrix

(x,y) posizione del pixel con il suo valore

Nel prossimo paragrafo verrà descritto il preprocessing (a partire dal grayscaling) di cui OpenCV si occupa.

OpenCV Python Library

Chapter 1

Image preprocessing

Grayscaling, adaptive thresholding, noise removal, histogram equalization: from theory to codes

Il *grayscaling* e l'*adaptive thresholding* sono rispettivamente la prima e l'ultima fase del preprocessing dell'immagine: il grayscaling, come si può evincere dalla parola stessa, converte l'immagine in scale di grigi; l'adaptive thresholding invece consente di ottenere un'immagine binaria, ovvero in bianco e nero. I processi intermedi, noise removal e histogram equalization, servono per migliorare ulteriormente l'immagine.(*)

Le funzioni adoperate sono le seguenti:

```
gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
```

per il ***grayscaling***, l'immagine viene fornita sottoforma di un numpy array, come descritto nella pagina precedente. **COLOR_BGR2GRAY**(**) è il *color space conversion code*; OpenCV fornisce anche conversioni in altre scale di colori.

```
blurred = cv.fastNlMeansDenoising(gray,50,7,21)
[ parametri: <>(src, h, templateWindowSize, searchWindowSize) ]
```

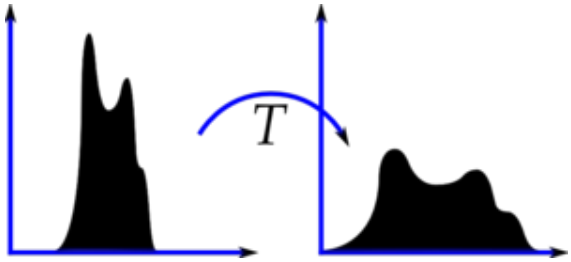
per la ***noise removal***;

- ***h*** è un parametro che regola la filter strenght: maggiore è il valore maggiore sarà la riduzione del rumore; di contro vi è la conseguente perdita di più dettagli dell'immagine.

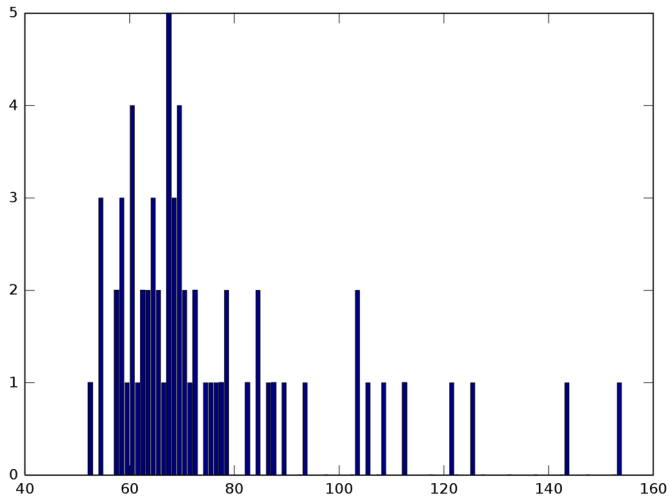
- ***templateWindowSize*** regola la dimensione della finestra per calcolare i “pesi” dei pixel.
- ***searchWindowSize*** regola la dimensione della finestra usata per calcolare il peso medio di ogni pixel. Tale parametro influenza linearmente le performance: maggiore sarà la dimensione della finestra, maggiore il tempo di denoising.

```
histed = cv.equalizeHist(thresh)
```

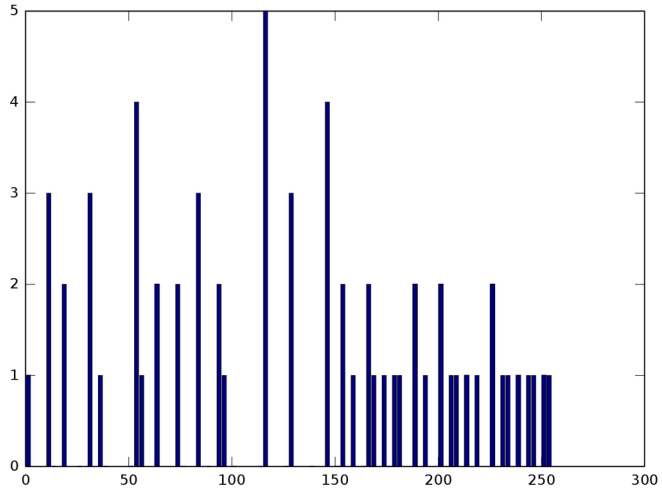
per l’***histogram equalization***. Principalmente questa funzione aumenta il contrasto dell’immagine al fine di migliorarne la leggibilità.



Solitamente, come mostrato nel grafico SX sopra, i pixel di una immagine tendono a concentrarsi su uno specifico *range* di valori. Nel caso siano luminose avremo valori alti, al contrario invece per le img con più ombra. Questa funzione di *OpenCV* porta al risultato mostrato nel grafico DX: i pixel sono omogeneamente concentrati su un più ampio *range*. Per ottenere un riscontro migliore sarebbe bene utilizzare immagini con un istogramma confinato su una particolare regione, come sopra.



Originale



Equalizzata

```
#first arg MUST BE a grayscaled img ; second arg : max pixel value
#third arg : adaptive threshold type (mean or gaussian) ; fourth arg:
threshold type (ONLY binary for adaptive)
#fifth arg: pixel block size ; 0 black 255 white (max value)
thresh = cv.adaptiveThreshold(blurred,255,cv.ADAPTIVE_THRESH_MEAN_C,
cv.THRESH_BINARY,17,1.8)
```

```
[ parametri:
<>(src,maxValue,adaptiveMethod,thresholdType,blockSize,C) ]
```

per l'***adaptive thresholding***. Questa funzione converte l'immagine da grayscale (ulteriormente manipolata dalle funzioni sopra riportate) a binaria, ottenendo una foto in bianco e nero.

Prima di spiegare l'adaptive è bene introdurre il *simple thresholding*: un metodo straight-forward che applica lo stesso valore soglia (threshold) ad ogni pixel. In questo caso, avendo usato *cv.THRESH_BINARY* come tipo di thresholding, se il valore del pixel è minore del threshold viene settato a 0 (nero), altrimenti al suo *maxValue* (bianco).

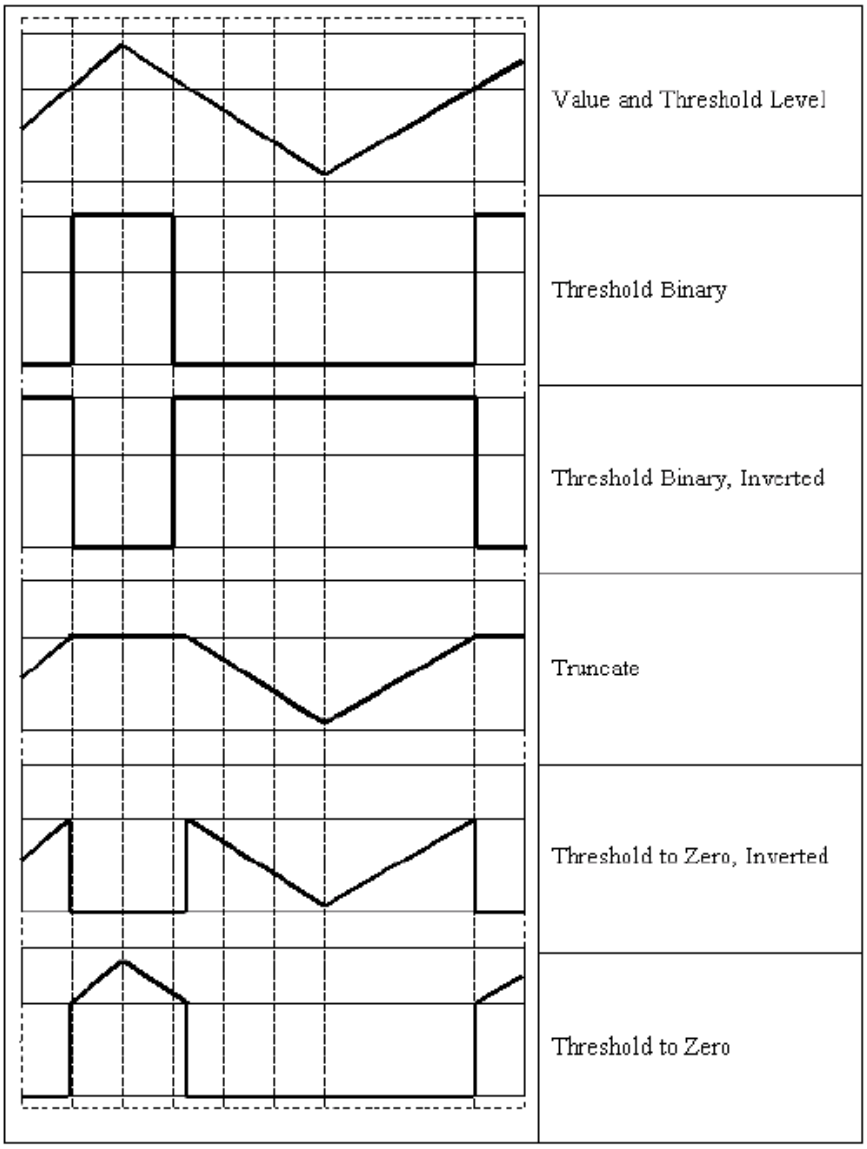
$$\texttt{dst}(x,y) = \begin{cases} \texttt{maxval} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$$

(***Simple thresholding*** di tipo *THRESH_BINARY,* semplice binarizzazione dell'immagine. Esistono altri tipi di thresholding, come spiegato sotto)

Se però l'immagine in questione risulta con differenti condizioni di luminosità su diverse regioni, questo tipo di binarizzazione potrebbe generare un risultato rumoroso; si ricorre pertanto all'*adaptive thresholding*, utilizzando comunque il tipo *THRESH_BINARY*, dove un algoritmo decide il valore soglia per ogni pixel basandosi su una piccola regione attorno ad esso. Si avranno threshold dedicati per ogni regione della foto.

$$dst(x,y) = \begin{cases} \texttt{maxValue} & \text{if } src(x,y) > T(x,y) \\ 0 & \text{otherwise} \end{cases}$$

(***Adaptive thresholding*** di tipo *THRESH_BINARY. Il valore di threshold ottenuto in questo caso non è un integer fisso* ***thresh*** *come nel caso del simple ma una funzione* `T(x,y)` *che dipende dalla regione attorno al pixel considerato)*

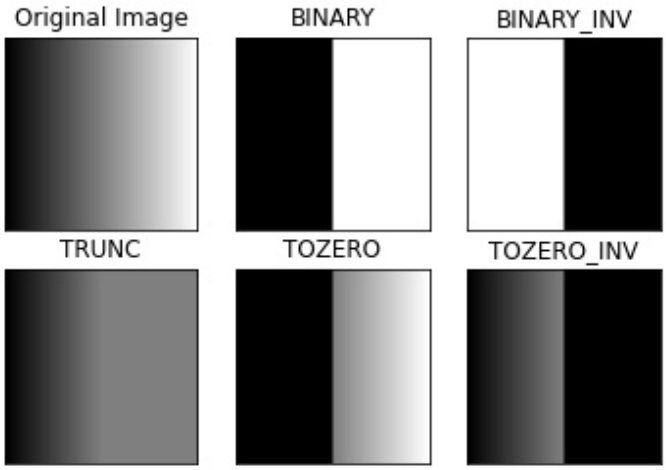


(Schema riassuntivo dei diversi tipi di thresholding presenti in OpenCV)

In sintesi: il tipo di thresholding (in questo caso binario) è indipendente dal metodo di scelta (simple o adaptive) del valore soglia.

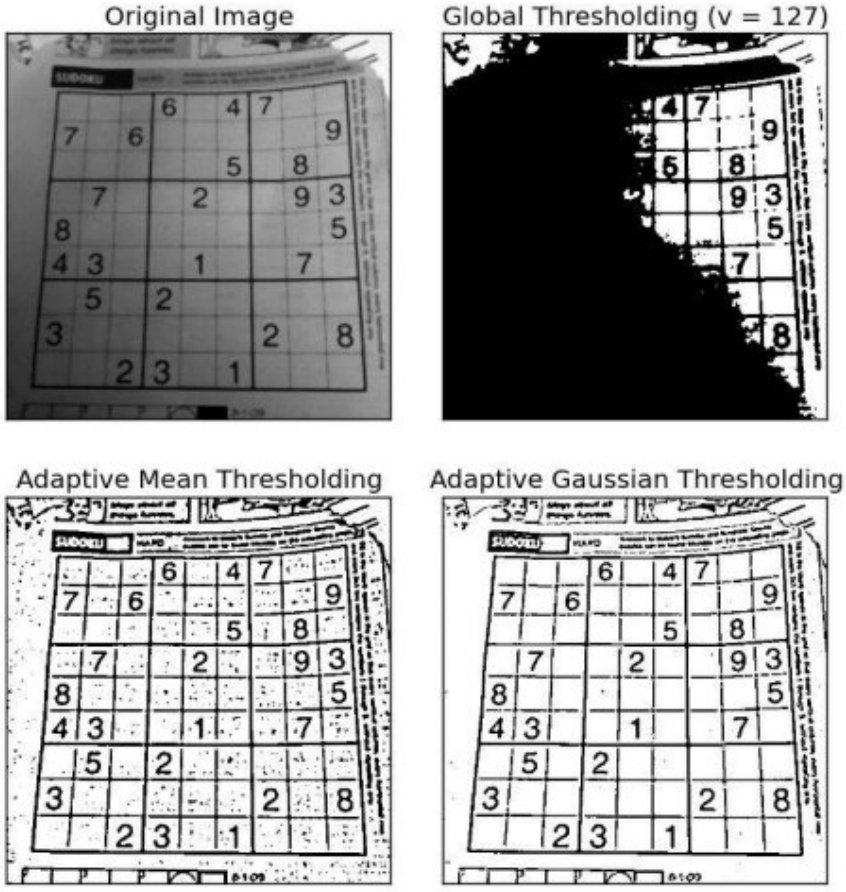
Riguardo i parametri presenti nella funzione:

- ***maxValue****,* il valore da settare ad un pixel se il suo valore è maggiore della soglia (come accennato in precedenza sul simple thresholding)
- ***C****,* una costante da sottrarre al valore di threshold trovato tramite i metodi scritti nel punto sotto.
- ***blockSize***, la dimensione della regione attorno al pixel cui calcolare il threshold.
- ***adaptiveMethod***, un integer che può assumere solo due valori: *ADAPTED_THRESH_MEAN_C* oppure ADAPTIVE_THRESH_GAUSSIAN_C. In questo progetto è stato scelto il primo: il valore di threshold di un pixel viene scelto facendo la media tra i pixel della sua regione per poi sottrarre di una costante C. L'altra versione usa invece una somma gaussiana invece della media.
- ***thresholdType***, un integer che specifica il tipo di threshold scelto. Ve ne sono diversi in questa libreria:



Come risultato finale serve una immagine in puro bianco e nero, di conseguenza si è scelto il tipo *cv.THRESH_BINARY.*

In OpenCV è presente inoltre un altro tipo di binarizzazione oltre alla simple e all'adaptive, cioè quella di *Otsu*: segue la scia della simple, solo che il valore soglia non viene scelto manualmente ma determinato in maniera automatica.



Questa immagine esplica il motivo per cui si è adottato il *mean thresholding* e non il *gaussian*: sebbene il risultato ottenuto sia meno rumoroso nel secondo caso, il sistema OCR rileva quei "punti" come caratteri di interpunzione, andando a sporcare notevolmente i dati estratti.

(*) Non è noto se eseguendo i processi intermedi successivamente all’adaptive thresholding si possano ottenere immagini disturbate, l’ordine delle operazioni è stato deciso solo in base al rapporto tra il tempo di esecuzione e la qualità finale dell’immagine

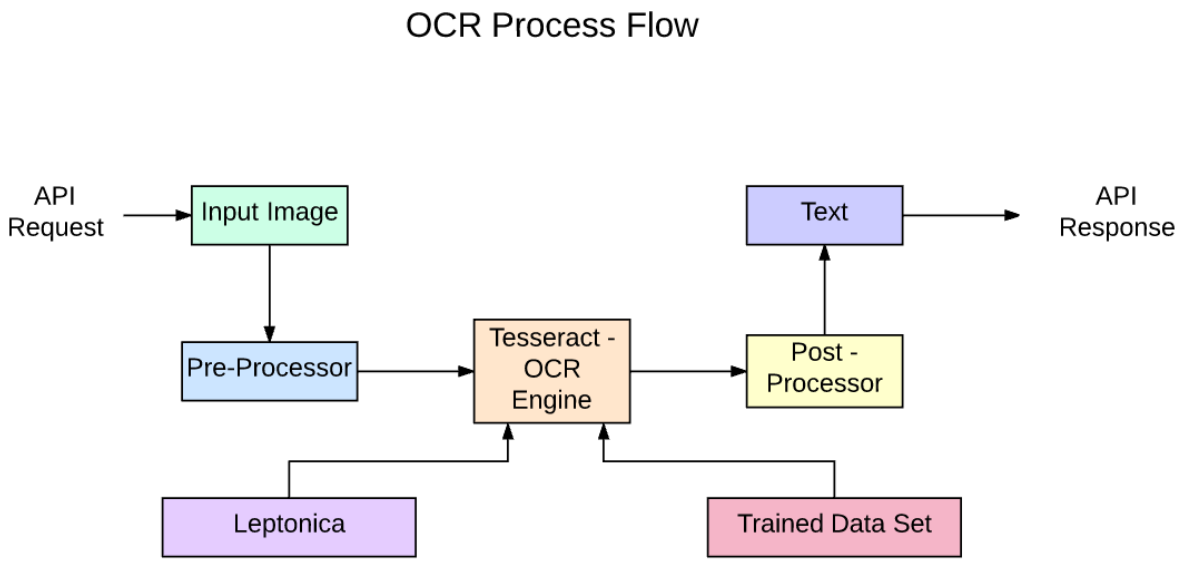
(**)BGR è lo stesso spazio di colori normalmente denominato RGB, con l’ordine delle aree dei colori invertite

Una volta aver terminato il preprocessamento dell’immagine, riuscendo ad ottenere un risultato in binario (B/W) e ad alto contrasto, arriva il turno di PyTesseract che, settato con certi parametri spiegati in seguito, sfrutterà il suo algoritmo per estrarre per intero tutto ciò che riguarda i testi. I suoi compiti saranno infatti quelli di *segmentation* e *feature extraction*. Il post processing segue invece un algoritmo scritto da me con l’aiuto di semplici *regex* (espressioni regolari), al fine di suddividere le informazioni appena estratte e prelevare solamente quelle utili al JSON finale.

PyTesseract library

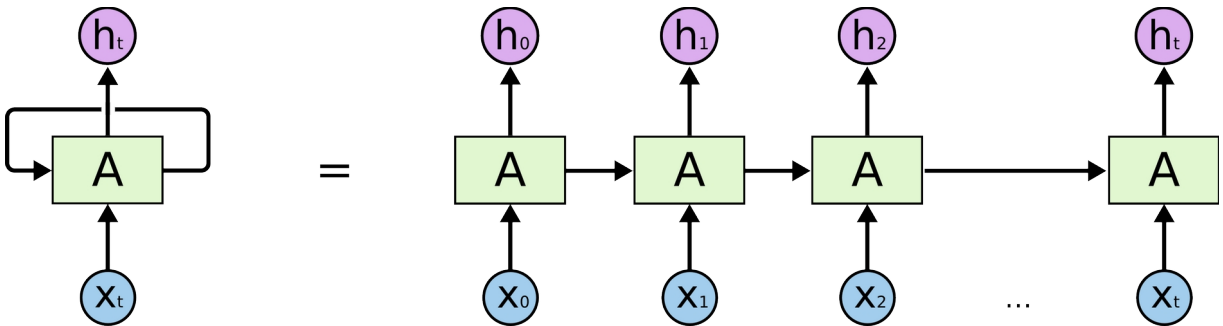
Chapter 2

Segmentation and feature extraction: functions between pre and post processing



PyTesseract (Python-tesseract) è un tool **OCR** capace di “leggere” il testo contenuto nelle immagini, con notevole supporto ad una larga varietà di lingue. É un *wrapper* di *Google’s Tesseract-OCR-Engine.*

La versione *4.00* include una nuova rete neurale configurata come text line recognizer. Per il riconoscimento di un singolo carattere all’interno di una foto viene usata la **CNN** (*Convolutional Neural Network*); per interi testi, che non sono altro che sequenze di caratteri di lunghezza arbitraria, vengono utilizzate le **RNN** (*Recurrent Neural Network*), dove i *neuroni* sono collegati in loop tra loro.



(*In figura:* A *è il chunk della rete neurale che guarda ad un input* x_t, *mentre* h_t *è l'output. Il loop consente il passaggio dell'informazione ottenuta da uno stato all'altro)*

Questi loop consentono di preservare gli stati precedenti ad un dato stato x. Mi spiego meglio: le tradizionali reti neurali pensano solamente "from scratch", senza alcun riferimento agli istanti temporali passati; le RNN invece restituiscono un output basato anche sulla memoria del passato.

Trovano pertanto impiego in campi come *speech recognition*, *language modeling*, *translation* ed ovviamente ***l'OCR***. Dalla figura sopra è inoltre facilmente intuibile come queste reti siano strettamente collegate ai concetti di liste e sequenze nel campo delle strutture dati. Dunque in sintesi: collegano l'informazione attuale con la precedente.

Esempio pratico che consente di capire meglio come si muove sull'OCR: una RNN può essere capace di predire la prossima parola basandosi sulla frase in precedenza ricevuta. Per gli OCR viene dunque eseguita una fase di training, e diventa pertanto più una questione di confronto (tramite una funzione di probabilità) che di completamento.

Un esempio pratico di RNN utilizzata per l'OCR è la **LSTM** (*Long Short Term Memory*). Sono speciali reti ricorrenti con l'obiettivo di memorizzare l'informazione per lunghi periodi di tempo. Le classiche RNN si mantengono su un arco temporale ristretto, dunque si tratta di eventi comunque recenti. Le LSTM infine, pur essendo efficienti nel lavorare su sequenze di dati, sono lente quando il numero di stati è troppo elevato.

Implementata su PyTesseract, una LSTM suddivide l'immagine in input in box (rettangoli). Il box i-esimo rappresenta la i-esima linea di testo data in input alla rete. Ovviamente più questi modelli LSTM vengono trainati, specialmente con font di tipi diversi, più si acquisiscono migliori performance.

Nel caso di PyOCRBusinesscard, il metodo invocato dalla libreria PyTesseract è *image_to_data*. Esso consente la collezione di parole racchiuse in box.

Di seguito la spiegazioni dei parametri del metodo:

```
image_to_data(img,output_type=Output.DICT,config='—psm 11 -c
tessedit_do_invert=0',lang='ita',)
```

- ***img*** è l'immagine ricevuta in input, preprocessata tramite i metodi di OpenCV, come illustrato nel capitolo precedente.
- ***output_type*** indica il formato in cui il testo estratto viene restituito. In questo caso, per avere una manipolazione facilitata dello stesso, ho scelto un dizionario tramite il parametro *Output.DICT*.
- ***config*** contiene flag di configurazione aggiuntivi. Con *--psm 11* si specifica la *Page Segmentation Mode*, ovvero la modalità con la quale il metodo dovrà scansionare, sezionare ed estrarre le parole (appunto il processo di segmentazione). In questo caso è stata scelta una modalità di "testo sparso", al fine di trovare quanto più possibile senza un ordine particolare. L'ordine infatti non è necessario, in quanto le informazioni utili vengono ricavate a valle tramite *regex.* Sono comunque presenti altri tipi di segmentazione della pagina/immagine: la si può trattare come un unico carattere, un'unica parola, un'unica riga di testo; alternativamente si può scegliere la modalità automatica.
- ***lang*** esplicita la lingua. Sono possibili selezioni multiple e per ognuna di essa è necessario scaricare un modello di training reperibile dall'account github del progetto PyTesseract.

JSON Building Algorithm

Chapter 3

Building process of the extracted PyTesseract dictionary through simple regular expressions (regex) and others controls

L'ultimo step per la creazione finale del JSON con le informazioni ricavate tramite PyTesseract consiste semplicemente nel coglierle tramite espressioni regolari e altri controlli sulle parole e buildare un nuovo dizionario che sarà poi dumpato in JSON.

Di seguito il codice.

```
def regexFind(file):
     mailPattern = '\S+@\S+'
     namePattern = "([A-Z][a-z]*)([\\s\\'-][A-Z][a-z]*)*"

     card = {"Type": "itemnote_create", "Context": "OP",
     "note": "", "description": "", "type": "visitor","station":""}

     #tmp
     ns = {"name": "","surname":""}
     opt = {"piva":"","phone":"","website":""}

     #for more accuracy with names
     titles = ['Dott.','Ing.','Dott.ssa','Avv.']

     file.seek(0)
     firstLine = file.readline()
     print(firstLine)

     #splitting in words
     arr = firstLine.split()
```

```
#finding using titles
for i in range(len(arr)):

     if re.search("P.IVA",arr[i]) or re.search("P IVA",arr[i])
     or re.search("PIVA",arr[i]):

          opt['piva'] = arr[i+1]

          #searching for +39
          if arr[i]=="+39":
               if arr[i+1]==" " or len(arr[i+1])<6:
                    opt['phone'] = "+39"+arr[i+2]
               else:
                    opt['phone'] = "+39"+arr[i+1]

          if re.search("www",arr[i],re.IGNORECASE) or
          re.search("https",arr[i],re.IGNORECASE) or
          re.search("http",arr[i],re.IGNORECASE):
               opt['website'] = arr[i].lower()

          for x in range(len(titles)):
               if re.search(titles[x],arr[i]):
                    ns['name'] = arr[i+1]

                    if(len(arr[i+2])<=4):
                         ns['surname'] = arr[i+2] + " " +
                         arr[i+3]
                    else:
                         ns['surname'] = arr[i+2]
               x=x+1
          i=i+1

          # ( m(n) : array[] ) -> m(n)[0] full mat
          try:
               m=re.search(mailPattern,firstLine)
          except TypeError:
               print("Mail not found.")
               exit()
```

```
#mail
card['note'] = m[0]
card['note'] = card['note'].replace(':','.')

#company by mail
card['station'] = card['note'].split("@")
[1].split(".")[0].capitalize()

#if it did not find name by titles
if ns['name']=="" or ns['surname']=="":
#regex matching between name pattern and first line
     n=re.findall(namePattern,firstLine)

     #new lists
     firstTElem = []
     secondTElem = []

     #for loop over name matches
     #splitting dict into two separate arrays
     for a in n:
          if not a[0]=='' or a[0]==' ':
               firstTElem.append(a[0])
          else:
               firstTElem.append('X')

          if not a[1]=='' or a[1]==' ':
               secondTElem.append(a[1])
          else:
               secondTElem.append('X')

     stringName = card['note'].split("@")[0]
     i=0
```

```
            for x in firstTElem:
                if re.search(x,stringName,re.IGNORECASE) and
                len(x)>2:

                            ns['name'] = x.replace(" ","")
                            ns['surname'] =
                            secondTElem[i].replace(" ","")
                i=i+1

        card['description'] = ns['name']+" "+ns['surname']

        with open(ns['name']+ns['surname']+".json",'w') as fp:
            json.dump(card,fp)

        return card,ns['name']+ns['surname']+".json"
```

Il dizionario viene dumpato tramite il metodo *json.dump([...])* contenuto nella libreria *json* e poi utilizzato come valore di ritorno della funzione.

I commenti al codice esplicano ulteriormente come vengono processate le informazioni.

Back-end and front-end: Flask with Bootstrap template

Chapter 4

How to serve this project as a web application using Flask as a server for Bootstrap (front-end framework) templated pages

Finora è stato analizzato solamente l'aspetto "core" del progetto: ovvero come viene preprocessata l'immagine e come vengono estratte le informazioni per il build del JSON.

Ma come dare "in pasto" le immagini? Inizialmente sono stati effettuati dei test tramite due webcam: una GPIO in quanto il progetto veniva testato su di un Raspberry Pi 4 e una semplice webcam USB.

I risultati non sono stati soddisfacenti in termini di qualità, in quanto una webcam standard non è in grado di avere un focus tale da catturare nitidamente testi relativamente piccoli ad una classica distanza utente-pc.

La scelta finale è dunque ricaduta su un classico upload statico dell'immagine da una pagina HTML. E' stata anche ottimizzato il responsive per una corretta visualizzazione su dispositivi con diverse risoluzioni e aspect ratio.

Dato che il progetto è stato interamente scritto in Python, è stato scelto Flask come micro-framework web.

Di seguito il codice del source *app.py*, con relativa spiegazione della suddivisione delle pagine HTML e delle funzioni.

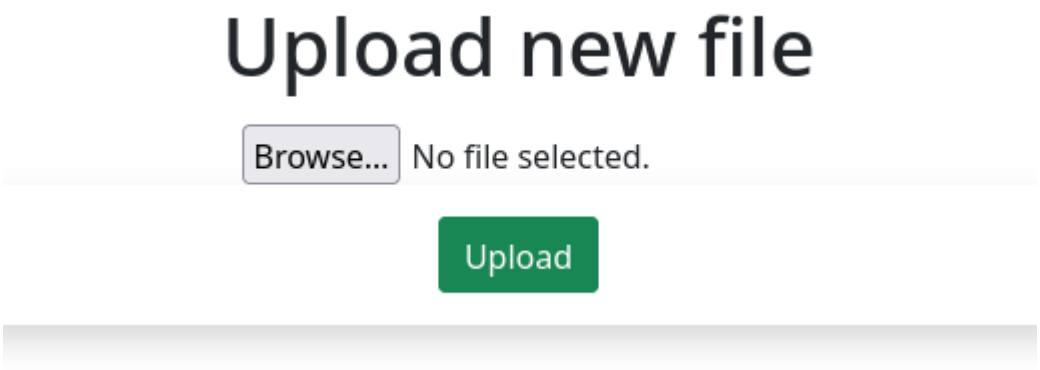
```
@app.route('/upload',methods=['GET','POST'])
def upload():

     if request.method == 'POST':
          file = request.files['file']
          if file.filename == '':
               return redirect(url_for('upload'))

          if file:
               filename = file.filename
               finalPath = os.path.join(app.config['UPFL'],filename)
               file.save(finalPath)

               name=bcModules.extraction(finalPath)

               fp=open(name,'r')
               fp.seek(0)
               f=fp.read()
```



In questa prima parte è stata programmata la funzione di upload sfruttando una POST request HTTP. Tale richiesta viene appunto scaturita dopo il click sul pulsante di upload. A meno che il file (l'immagine) non sia privo di nome, dunque un file nullo, esso verrà salvato in una path specifica.

```
res = make_response(f)
res.headers['Content-Type'] = 'application/json'
```

La funzione *bcModules.extraction(...)* ritorna il nome del .json generato, la cui “lettura” sarà passata come parametro alla *make_response(file)* che di fatto “crea” la risposta. Da notare anche *application/json* per specificare il contenuto.

Questo codice però implica l’avere il json finale in localhost, poco sensato per un uso pratico. Il codice che riporto qui sotto invece esegue la richiesta con relativa risposta finale verso un server in LAN, dove il .JSON verrà salvato.

```
url = "http://192.168.46.70:7478/..."
req = urllib.request.Request(url)
req.add_header('Content-Type','application/json')
jsondata = json.dumps(body)
jsonbytes = jsondata.encode('utf-8')

req.add_header('Authorization','Bearer xxx'
response = urllib.request.urlopen(req,jsonbytes)
print("Status code: %s" %response.getcode())
```

Il core dell’applicazione Flask è questo, il resto riguarda solo una banale gestione di pagine HTML e pertanto non è necessario da menzionare.

RedBot

Simple Node.JS script that forwards Redis data to a Telegram bot using APIs

Il funzionamento è basato sulle callback. Lo script si sottoscrive ad un canale di Redis e rimane continuamente in ascolto di nuovi messaggi. Ad ogni messaggio ricevuto scatterà una callback la quale invierà tale messaggio (o anche semplicemente una notifica di nuovi dati, come mostrato nello snippet esemplificativo sotto) ad ogni user che avrà avviato il relativo bot Telegram.

```
const subscriber = createClient();
subscriber.on("message", function(channel, message) {

console.log("Subscriber received message in channel '" + channel +
"': " + message);
bot.on('message', (msg) => {
bot.sendMessage(msg.chat.id,"new data on Redis");
});

});
subscriber.subscribe("redisChannel");
```

La funzione *createClient()* inizializza Redis tramite le sue API e consente la successiva sottoscrizione ad un suo canale.

```
function createClient() {
try {
const sock = process.env.REDIS_SOCK ?? '/usr/local/sock/redis.sock'
if (fs.existsSync(sock))
return redis.createClient(sock)
} catch (err) {
console.log(err)
}
```

```
console.log(`creating ${process.env['REDIS_HOST'] ||
'localhost'}:6379 client`)
return redis.createClient(6379, '192.168.46.30')
}
```

Il concept originario era quello di ricevere sul bot, in formato stringa, i JSON nel DB Redis provenienti da PyOCRBusinessCard (il progetto descritto prima), in quanto la POST HTTP request serviva proprio ad inviare il .JSON generato al database in LAN.
In realtà è però adattabile a qualunque tipo di dato che possa essere riportato in forma testuale (stringa, lista, dizionari, etc..), poiché è sufficiente passarlo come parametro alla funzione *bot.sendMessage(msg.chat.id, "<data string>").*

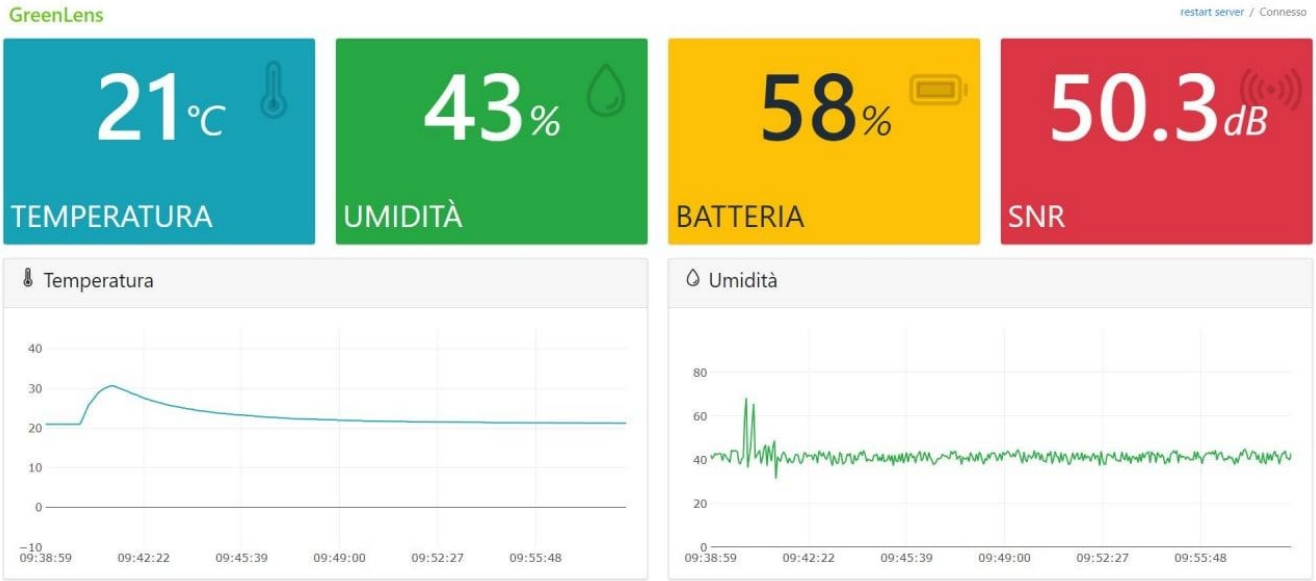
GreenLens

Web dashboard that shows plants info from sensors using Javascript and MQTT APIs

Questo progetto è stato realizzato in collaborazione con un altro studente tirocinante, Edoardo Campi, il quale si è occupato principalmente della parte hardware (sensori, LoRa, MQTT gateway, etc..). Il nome del progetto è invece stato ideato dal Responsabile Comunicazione dell'azienda, Simone Agresti. Il mio compito è stato quello di curare la visualizzazione del risultato finale, con una semplice dashboard HTML che gira su un server HTTP locale.

Una sintesi riguardo la sua parte del progetto, per meglio comprendere la mia: vi è il nodo (Arduino/LoRa), contenente un sensore e collegato direttamente alle piante che invia i relativi dati di temperatura, umidità, SNR e livello batteria al gateway (un Raspberry Pi). Pertanto: Arduino funge da publisher, il Raspberry da gateway e dunque da broker, mentre la dashboard sviluppata è il subscriber.

Quest'ultima è stata sviluppata da me, utilizzando un template grafico già esistente (da AdminLTE). Si tratta di un proof of concept, dunque non è stato realizzato un robusto web server ma, come accennato prima, un semplice server HTTP Python.



La connessione al broker MQTT (il gateway) è stata sviluppata in Javascript con l'ausilio della libreria Paho, non considerando eventuali controlli di sicurezza in quanto (visto che si tratta di un PoC). La dashboard si sottoscrive ad MQTT ed in tempo reale aggiorna i dati presenti sulla pagina, costruendo inoltre delle statistiche cronologiche come visibile nell'immagine sopra. A tal proposito, per plottare i dati su tali grafici, è stata utilizzata la libreria Plotly.

Lo snippet Javascript riportato sotto mostra la fase di connessione al broker MQTT.

```
client = new Paho.MQTT.Client(hostname,Number(port),clientId);

// callback su ogni messaggio arrivato
client.onMessageArrived = onMessageArrived;
//callback in caso di connessione persa
client.onConnectionLost = onConnectionLost;

function try_connect(){
client.connect({
onSuccess: onConnect,
onFailure: onConnectionLost,
keepAliveInterval: 10,
userName: username,
password: password,
});
n_try_connect=n_try_connect+1;
}
try_connect();

function Connected(){
console.log("connected");
}

function onConnect(){
client.subscribe(topic);
document.getElementById("stato").innerHTML = "Connesso";}
```

Di particolare interesse è la callback evidenziata, cioè *onMessageArrived,* di cui riporto il codice con successiva spiegazione.

```
function onMessageArrived(msg){

if(first_time==0){
Plotly.newPlot('tempCHART', [temperatura],layout_temp);
Plotly.newPlot('humyCHART',[humidity],layout_humy);
first_time=1
}


obj = JSON.parse(msg.payloadString);


temperatura.y.push(Number(obj.temp));
temperatura.x.push(obj.time);

humidity.y.push((Number(obj.humy)*100/1023));
humidity.x.push(obj.time);



if(obj.stato==true)
{
if(check_restart==false)
{document.getElementById("stato").innerHTML = "Connesso";}
}
else
{
document.getElementById("stato").innerHTML = "Restart in corso...";
check_restart=false;
}

// gestione modifica icona in base alla percentuale
if(Number(obj.Batt)<=5){
document.getElementById("batt-icon").setAttribute("name","battery-
dead-outline");
}
```

```
else if(Number(obj.Batt)<=50){
document.getElementById("batt-icon").setAttribute("name","battery-
half-outline");
}
else{
document.getElementById("batt-icon").setAttribute("name","battery-
full-outline");
}


Plotly.update('tempCHART', [temperatura],layout_temp);
Plotly.update('humyCHART',[humidity],layout_humy);


document.getElementById("temp").innerHTML = Math.round(obj.temp);
document.getElementById("humy").innerHTML =
(Math.round((Number(obj.humy)*100/1023))).toString(10);

document.getElementById("batt").innerHTML = obj.Batt;
document.getElementById("snr").innerHTML = obj.snr;
}
```

msg.payloadString è la stringa ricevuta dal gateway e contenente le informazioni da visualizzare e plottare sul grafico. La funzione *JSON.parse(…)* rende tale stringa un dizionario in modo da avere i campi accessibili separatamente (obj.temp, obj.humi, etc..).

Le quattro funzioni (provenienti dalla libreria Plotly) plottano sul grafico i dati di umidità e temperatura precedentemente ottenuti.

Le quattro *document.getElementById* servono invece a mostrare i valori in tempo reale delle quattro grandezze considerate.

Inoltre, i dati di temperatura e umidità vengono “salvati” temporaneamente in queste strutture dati, prima di essere sfruttate con Plotly.

```
var temperatura = {
x: [],
y: [],
type: 'scatter',
name: 'Blue',
marker: {
color: '#18a4b4',
size: 12
}
};


var humidity = {
x: [],
y: [],
type: 'scatter',
marker: {
color: "#28ac44",
size: 12
}
};
```

FluxBridge

Python gateway (server side, Flask) that receives data using POST HTTP requests and then forwards them to a Telegram bot making use of APIs

Questo script è collegato al progetto **GreenLens**, in quanto le grandezze misurate vengono pushate su di un database di tipo time series (InFlux) e poi manipolate attraverso la sua dashboard.

Nello specifico: su **InFlux** si setta un valore range di tali misurazioni al di fuori del quale si scaturisce un alert. La conseguenza di questo allarme è l'invio di una *HTTP POST* request al gateway; il contenuto è un dizionario prodotto da InFlux stesso con la grandezza, il valore misurato e altre informazioni aggiuntive, come mostrato in questa struttura dati:

```
msg = {"checkId": "",
"checkName": "",
"level":"",
"message": "",
"_field":0,
"sourceMeasurement": "",
"timestamp": 0
}
```

Dove *sourceMeasurement* rappresenta la misura, dunque temperatura o umidità nel caso di GreenLens.
La route di Flask di particolare importanza in questo script è la seguente:

```
@app.route('/notify',methods=['POST'])
def notify():

if request.method == 'POST':
     print("notification received",flush=True)
     msg["checkId"] = request.json["_check_id"]
     msg["checkName"] = request.json["_check_name"]
     msg["level"] = request.json["_level"]
```

```
        msg["message"] = request.json["_message"]
        msg["sourceMeasurement"] = request.json["_source_measurement"]
        msg["timestamp"] = request.json["_status_timestamp"]

    return {'result':"OK 201"}
```

Su InFlux viene settato come *endpoint* per le POST request l'IP di tale server; dato che il tutto è stato svolto in locale, l'indirizzo assumerà una forma del tipo *192.168.XXX.XXX/notify.*
Il passo successivo è salvare in una struttura dati temporanea *msg* i campi del JSON passato con la POST che servano al nostro scopo.

Adesso veniamo ad analizzare lo sviluppo della parte riguardante il bot Telegram che si occuperà di notificare ogni *nuovo* e *utile* dato.
I moduli necessari dalla libreria *telegram.ext* sono i seguenti:

```
from telegram.ext import Updater, updater, Filters, CommandHandler
```

Il bot è composto da 3 comandi: start, set del timer e stop

```
updater = Updater(token)

updater.dispatcher.add_handler(CommandHandler("start",start))
updater.dispatcher.add_handler(CommandHandler("set",callback_timer))
updater.dispatcher.add_handler(CommandHandler("stop",callback_stop))
```

che vengono inizializzati utilizzando la classe *CommandHandler.*
Il comando di *start* ritorna un messaggio per il *set del timer*.

```
def start(update,context):
    update.message.reply_text("Usage: /set <seconds> to set a
                              timer")
    return
```

Seguendo un approccio bottom-up, passo a descrivere prima la funzione chiave, quella che fa pushare sul bot le notifiche, e successivamente l'handler di essa.

```
def pushToBot(context):
     job = context.job

     if(msg["sourceMeasurement"]!=msgTmp["sourceMeasurement"] or
        msg["timestamp"]!=msgTmp["timestamp"]):

          if(msg["level"]=="crit"):
               text = emoji.emojize(':red_circle: ')+"CRITICAL LEVEL
                                    | "+list(msg)[3]+":
                                    "+str(msg[list(msg)[3]])

          if(msg["level"]=="warn"):
               text = emoji.emojize(':warning: ')+"WARNING LEVEL |
                                   "+msg[list(msg)[3]]+":
                                   "+str(msg[list(msg)[3]])

     context.bot.send_message(context.job.context,text=text+"\n\
                              n"+json.dumps(msg))

     msgTmp["sourceMeasurement"] = msg["sourceMeasurement"]
     msgTmp["timestamp"] = msg["timestamp"]

     return
```

Questa funzione è basata sulla classe *telegram.ext.Job*, molto comoda da utilizzare per la schedulazione dei processi. In realtà, quasi tutto lo script è basato sui *Job,* poiché grazie ad essi diviene semplice lavorare con le callback.
Inizialmente viene eseguito un controllo per evitare la ricezione di duplicati sul bot, grazie al salvataggio su una seconda struttura dati temporanea *msgTmp.*

context.bot.send_message(...) consente la ricezione del contenuto, sul quale viene prima eseguito l'encoding con *json.dumps(msg)*.

Tornando ai comandi iniziali: dopo lo */start* *d*igitando */set* sul bot partirà la seguente callback:

```
def callback_timer(update,context):
     chat_id = update.message.chat_id
     try:
     # args[0] should contain the time for the timer in seconds
          interval = int(context.args[0])
          if interval < 0:
               update.message.reply_text('Please type n>=0.')
               return

     job_removed = removeJobIfExists(str(chat_id), context)
     context.job_queue.run_repeating(pushToBot, interval,
                                     context=chat_id,
                                    name=str(chat_id))

     text = 'Timer successfully set! Now '+str(interval)+' s.'

     if job_removed:
          text += ' Old one was removed.'
          update.message.reply_text(text)

     except (IndexError, ValueError):
          update.message.reply_text('Usage: /set <seconds>')
```

Tralasciando il controllo sul parametro del timer, il core di questa funzione è la chiamata a *context.job_queue.run_repeating*. Essa prende come parametri la funzione *pushToBot* (descritta in precedenza) e *interval*: lo scopo è quello di runnarla ogni *interval* secondi.
Sostanzialmente, con *interval=0* si ottengono notifiche/alert/messaggi push indipendentemente dalla frequenza di rilascio degli alert da parte di *InFlux*. In ogni caso, se tale frequenza è maggiore dell'inverso di *interval*, nessun messaggio viene perso.

L’altra callback è una funzione di servizio che elimina l’attuale timer settato, killando il *job* tramite *removeJobIfExists(...)*.

```
def callback_stop(update,context):
     chat_id = update.message.chat_id
     job_removed = removeJobIfExists(str(chat_id),context)
     text = 'Stopped. Job removed.'

     if not job_removed :
          text = 'No active timer'

     update.message.reply_text(text)

     return



def removeJobIfExists(name,context):
     currentJobs = context.job_queue.get_jobs_by_name(name)

     if not currentJobs:
          return False

     for job in currentJobs:
          job.schedule_removal()

     return True
```

jPianoPush

Pushing data to cloud from a log read dinamically from a file in continous writing

Per simulare un logfile aziendale *test.trace* in continua scrittura, ho deciso di sviluppare uno script Python *writing.py* che generi stringhe su file con i seguenti pattern:

```
[com.aec.server.ServiceProcessor][<-- REQUEST NAME: X -
PARAMS:id=1137084;machine=F1445;badge=0007554933;pressure=17;pressure
1=19;speed=0.18;speed1=0.18;temperature=25;temperature1=25]"

[com.aec.server.ServiceProcessor][--> ANSWER TO: spc_session_change
TIME(ms): X RESULT: 000000 - Executed - Servizio eseguito]"

[com.aec.server.ServiceProcessor][Enqueued service spc_session_change
for processing (size=X)]"

ServiceProcessor: [LOG] --> ANSWER TO: machinedata_insert TIME(ms)
[process - elapsed]: X - X2 RESULT: 00000000 - Executed - Servizio
eseguito"
```

dove i dati X e X_1 vengono generati con una distribuzione gaussiana tramite questa funzione:

```
import random

def gaussianCurve(mu,sigma,bottom,top):
     n = random.gauss(mu,sigma)

     while(bottom <= n <= top ) == False:
          n = random.gauss(mu,sigma)

     return n
```

dove:

- *mu* è la mediana
- *sigma* la deviazione standard
- *bottom* e *top* sono i valori di range della curva

Per quanto riguarda invece la consumazione del dato, è presente sia una versione in Python che in Java.

Python

La lettura dinamica del file avviene tramite la funzione *readnext(file,lastseen,globalseen):*

```
while True:
     lastseen = readnext("follow.txt",lastseen,globalseen)
```

Con i parametri settati a:

```
lastseen=0
globalseen=0
```

```
def readnext(filename,lastseen,globalseen):

     if (lastseen>=globalseen):
          globalseen=lastseen

          with open(filename) as file:
               file.seek(lastseen)

          for line in file:
               events(line)

          return file.tell()
```

- *lastseen* indica la riga attuale in cui il puntatore è posizionato, inizialmente 0. Non è necessario però dichiararlo per forza al di fuori della chiamata alla funzione *readnext(...)* in quanto diventa utile solo quando ritornato dalla *file.tell()*.

- *Globalseen* è una variabile di appoggio con l'utilità di permettere di capire quando bisogna leggere nuovamente; lo script infatti non legge se il puntatore alla chiamata successiva si trova nella stessa riga, condizione per cui è palese *globalseen<=lastseen*.

Ogni nuova linea viene passata ad *events(line)* che ne analizza il contenuto tramite la funzione di libreria regex *re.match(pattern,line)*. Nel caso di match tra riga e pattern i valori utili vengono prelevati tramite la funzione *finalString.group(n).*

La funzione di pushing sul cloud riguarda solamente delle POST request generabili con la libreria *requests:*

```
res = requests.post(qUrl,headers=headers,data=json.dumps(dict))
```

Le strutture dati pushate sono le seguenti:

```
queueing = {
"result": "push",
"data": { "size": -1, "service": '' },
"name": "service",
"source": "francesco_test",
"timestamp": -1
}


exiting = {
"result": 'pull',
"data": { "service": ''},
"name": 'queue',
"source": 'francesco_test',
"timestamp": 0
}
```

```
ending = {
"result": '',
"data": {
"service": 'machinedata_insert',
"processtime": 0,
"queuetime": 0 },
"name": '',
"source": 'francesco_test',
"timestamp": 0
}
```

Java

Nella versione Java la lettura dinamica del file avviene tramite la classe *BufferedReader*. Si comporta efficacemente in quanto minimizza il numero di operazioni I/O conservando i chunk di caratteri letti in un buffer interno: quando il buffer si riempie, il *Reader* leggerà direttamente da esso e non dallo stream. Inoltre, con tale classe viene meno il bisogno di tenere traccia della posizione del puntatore come in Python poiché è implicito.

```
BufferedReader br = new BufferedReader(new FileReader("test.trace"));

String currentLine = null;

while (true) {
     if ((currentLine = follow.readLine()) != null) {
          events(currentLine);
          continue;
     }

     try {
          Thread.sleep(1);
     } catch (InterruptedException e) {
          Thread.currentThread().interrupt();
          break;
     }
```

La funzione di pushing è stata invece scritta sulla base delle classi *HttpClient* e *HttpRequest*.

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder(URI.create(url))
                      .header("content-type","application/json")
                      .header("authorization",auth)
                      .POST(HttpRequest.BodyPublishers.ofString(json.
                            toString()))
                      .build();

HttpResponse<String> res = client.send(request,
HttpResponse.BodyHandlers.ofString());
```

Le espressioni regolari sono gestite dalle classi *Pattern* e *Matcher*, quindi nel caso della riga riguardante la struttura dati *queueing*:

```
Pattern pQueue = Pattern.compile(queue);
Matcher mQueue = pQueue.matcher(line);
```

Le strutture dati sono state scritte utilizzando il tipo *HashMap<String,Object>,* quindi per esempio:

```
HashMap<String,Object> queueing = new HashMap<>();
```