



SAPIENZA
UNIVERSITÀ DI ROMA

Miglioramento dell'adattività del metodo IS_{IS} e integrazione con codifica tramite operatori

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di Laurea in Informatica

Candidato

Eduardo Rinaldi

Matricola 1797800

Eduardo Rinaldi

Relatore

Prof.ssa Maria De Marsico *Maria De Marsico*

Anno Accademico 2019/2020

Sommario

Il presente lavoro di tesi ha come obiettivo quello di implementare un sistema di localizzazione dell’iride da utilizzare come base per le fasi successive. Il sistema è stato realizzato utilizzando come metodo di segmentazione *ISIS v.2* [12] e per l’estrazione delle feature e la codifica delle iridi la combinazione degli operatori **LBP** e **Spatiogram** [11].

Per garantire un’ottima efficienza l’implementazione di questo sistema è stata effettuata in C++ (v. 17) utilizzando la libreria di processing di immagini OpenCV. Successivamente si è proseguito **migliorando l’adattività** del metodo di segmentazione andando ad implementare un modulo che decidesse, sulla base di diversi controlli, se è necessario impiegare l’uso di un classificatore di occhi per individuare l’area di interesse o se l’occhio occupa già completamente l’immagine.

Indice

1	Introduzione	1
1.1	Cenni storici della biometria	1
1.2	Riconoscimento biometrico	1
2	Iride come base di un sistema biometrico	3
2.1	Fasi per il riconoscimento	3
2.2	Panoramica sulla relazione	4
2.3	Tecnologie utilizzate	4
2.3.1	C++	4
2.3.2	OpenCV	4
2.3.3	Python e MySQL	4
2.3.4	Dataset	5
3	Integrazione di <i>IS_{IS}</i> con il codice di codifica	6
3.1	Funzionamento di <i>IS_{IS}</i> v.2	6
3.1.1	Preprocessing	6
3.1.2	Segmentazione	8
3.1.3	Normalizzazione	9
3.2	Funzionamento del modulo di codifica e matching	10
3.2.1	Codifica	10
3.2.2	Matching	11
3.2.3	Performance del riconoscimento tramite operatori	12
3.3	Risultati dell'integrazione dei due codici	13
4	Miglioramento di <i>IS_{IS}</i> v.2	15
4.1	Nuova implementazione del filtro di posterizzazione	15
4.2	Ottimizzazione ricerca limbo e pupilla	17
4.3	Soluzioni parziali	18
4.3.1	Utilizzo di un filtro di blurring	18
4.3.2	Miglioramento individuazione dei riflessi	19
4.4	Tentativi falliti	20
4.4.1	Positioning di un cerchio	20
5	Lavoro sugli istogrammi	21
5.1	L'idea	21
5.2	Primo approccio: RGB	22
5.3	Passaggio allo spettro di colori HSV	22
5.3.1	Quantizzazione	24
5.3.2	Distribuzione delle fasce	25
5.3.3	Primi risultati	26
5.4	Introduzione del dataset Ubiris v2	28

5.5	Analisi massiva	30
5.5.1	Fascia del giallo e del verde	30
5.5.2	Fascia del blu e del viola	31
5.5.3	Fascia dell'arancione e del rosso	32
5.5.4	Conclusioni della prima analisi massiva	33
5.6	Individuazione del discriminante	33
5.7	Riprogettazione di <i>needClassifier</i>	34
5.7.1	Definizione e testing del primo controllo	35
5.7.2	Migliorie apportate a C ₁	35
5.7.3	Introduzione del secondo controllo	38
5.7.4	Risultati ottenuti dalla funzione <i>needClassifier</i>	45
6	Conclusioni	46
6.1	Sviluppi futuri	46
Bibliografia		47

Capitolo 1

Introduzione

Viviamo in un mondo nel quale esistono numerose applicazioni che erogano servizi generando migliaia di dati sensibili e ciò porta alla necessità di implementare ed utilizzare sistemi di autenticazione all'avanguardia alternativi alle classiche password, andando ad esempio ad utilizzare tratti biometrici per identificare un individuo. Il riconoscimento dell'iride fa parte di questa categoria di sistemi.

1.1 Cenni storici della biometria

Il primo metodo d'identificazione biometrico fu sviluppato a Parigi nel XIX secolo da Alphonse Bertillon, il quale fu assunto come fotografo di servizio per la prefettura di Parigi.

Il suo sistema, oggi conosciuto come Bertillonage, si basa sull'idea che l'ossatura umana è differente per ogni persona e che questa rimane invariata dopo il ventunesimo anno di età. Consiste quindi nell'annotare delle foto di riconoscimento con una serie di misurazioni di elementi del corpo, come cranio, arti, dita, piedi, naso e orecchie. Inizialmente questo sistema si diffuse in tutta l'Europa continentale, ma successivamente, data la grande precisione richiesta nell'essere utilizzato e data la scoperta di un nuovo sistema d'identificazione, ovvero quello dell'impronta digitale, fu presto abbandonato.

Nel 1892 Francis Galton arrivò a creare un sistema di identificazione attraverso l'impronta digitale [5] basandosi sull'idea che queste fossero diverse per ogni individuo.

Al giorno d'oggi sono noti anche altri sistemi che si basano su caratteristiche fisiche come l'iride o su caratteristiche comportamentali come la voce o la camminata.

1.2 Riconoscimento biometrico

Le caratteristiche utilizzate per effettuare un riconoscimento biometrico possono essere:

- **Caratteristiche fisiologiche:** non cambiano nel tempo. Fanno parte di questa categoria: l'iride, la retina, le impronte digitali, la fisionomia del volto, la fisionomia dell'orecchio e il DNA.
- **Caratteristiche comportamentali:** possono variare nel tempo e possono essere condizionate da fattori esterni. Fanno parte di questa categoria: l'impronta vocale, la scrittura e l'andamento della camminata.

Ognuna di queste per essere utilizzata in un sistema di riconoscimento deve essere:

- **Universale:** chiunque deve possedere questa caratteristica
- **Unica:** deve permettere di distinguere un individuo da un altro
- **Permanente:** non deve cambiare nel tempo
- **Misurabile:** deve poter essere misurata quantitativamente

Capitolo 2

Iride come base di un sistema biometrico

L'iride è una caratteristica fisiologica che rispetta in pieno le proprietà elencate nel capitolo precedente e la sua acquisizione non è invasiva, proprio per questo viene spesso utilizzata come caratteristica per un sistema di riconoscimento biometrico.

2.1 Fasi per il riconoscimento

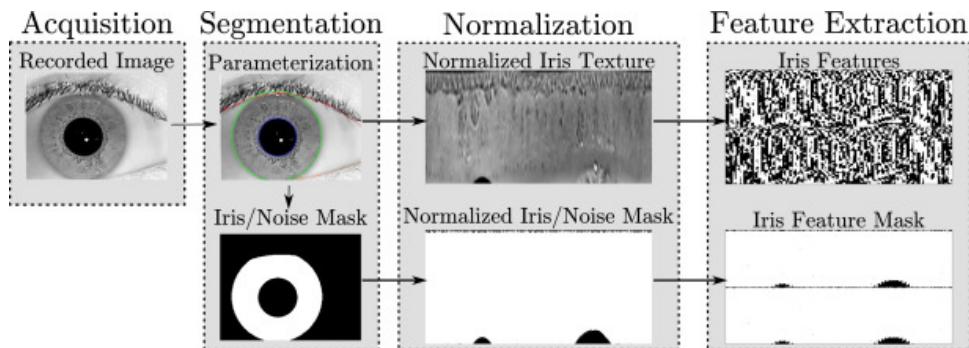


Figura 2.1. Pipeline per il riconoscimento dell'iride

In genere un sistema di riconoscimento dell'iride è composto dalle seguenti fasi:

1. **Acquisizione:** viene catturata l'immagine dell'iride che verrà elaborata successivamente. La cattura di quest'ultima può essere effettuata o nello spettro visibile dove l'iride rivela dei pattern molto casuali, o nello spettro infrarosso nel quale anche delle iridi molto scure hanno una tessitura più visibile.
2. **Preprocessing:** vengono applicati una serie di filtri o modifiche all'immagine in modo tale da migliorare la segmentazione che avverrà nella fase successiva. Esempi di modifiche applicabili sono ad esempio l'individuazione di un'area di interesse più precisa o l'applicazione di un filtro per la rimozione dei riflessi.
3. **Segmentazione:** viene localizzata l'iride andando a definire un cerchio per il limbo ed un altro per la pupilla.

4. **Normalizzazione:** in questa fase viene fatto l'unwrapping dell'iride localizzata precedentemente portandola poi in coordinate polari così da poterci lavorare più facilmente.
5. **Codifica:** vengono estratte le caratteristiche dell'iride, anche chiamate feature, attraverso l'utilizzo e la combinazione di operatori come LBP o Spatiogram.
6. **Matching:** viene effettuato un confronto tra le immagini prese in esame attraverso una metrica di similarità o di distanza.

2.2 Panoramica sulla relazione

La prima parte di lavoro riguarda l'integrazione del metodo di segmentazione ***IS_{IS}*** v2 [12] con il metodo di codifica e matching attraverso la **combinazione degli operatori LBP e Spatiogram** [11].

Sia per la segmentazione che per la codifica si è partiti da implementazioni già presenti; quella per *IS_{IS}* è stata scritta nel linguaggio Java, quindi si è pensato che lavorando più a basso livello, utilizzando un linguaggio come il C++, si potessero ottenere così delle sostanziali ottimizzazioni.

Nella nuova implementazione sono stati apportati dei miglioramenti importanti del preprocessing e delle ottimizzazioni dei costi computazionali su molte funzioni presenti.

Per quanto riguarda il codice di codifica e matching l'implementazione precedente era in C, utilizzando C++ è stata fatta un'operazione di "modernizzazione" del codice utilizzando nuove funzionalità fornite dalle nuove versioni del linguaggio, come ad esempio la possibilità di poter creare delle classi.

2.3 Tecnologie utilizzate

2.3.1 C++

Per svolgere il seguente lavoro di tirocinio come già detto è stato utilizzato il linguaggio **C++** per l'implementazione del codice di riconoscimento, in particolar modo è stata utilizzata la versione moderna (versione 17) di questo linguaggio, la quale ha facilitato notevolmente la scrittura del codice.

2.3.2 OpenCV

Come libreria principale per processare le immagini è stata utilizzata **OpenCV**, in modo tale da avere anche un grande supporto dato che è una libreria gratuita molto diffusa nel mondo della Computer Vision ed inoltre è la libreria utilizzata nelle implementazioni precedenti sia nel codice di segmentazione che in quello di codifica.

2.3.3 Python e MySQL

Per quanto riguarda la fase di testing del software è stato utilizzato il linguaggio **Python** (versione 3) per creare con estrema facilità degli script per fare test massivi sui dataset e per salvare i risultati su una base di dati in combinazione con **MySQL**; quest'ultimo è stato scelto per le sue potenzialità nell'esprimere interrogazioni sui dati.

2.3.4 Dataset

I dataset utilizzati per il testing sono:

1. **Ubiris** (versione 2) [7]: sono presenti più di 10000 immagini catturate su poco più di 500 occhi diversi. Il dataset fornisce informazioni riguardo la distanza di cattura e le condizioni in cui sono state scattate.
2. **Utiris**: sono presenti circa 800 immagini catturate su 79 soggetti diversi. Il dataset specifica per ognuno di essi se l'immagine raffigura il suo occhio destro o sinistro.

Capitolo 3

Integrazione di *IS_{IS}* con il codice di codifica

3.1 Funzionamento di *IS_{IS}* v.2

Come già detto, *IS_{IS}* implementa algoritmi per le fasi di preprocessing, segmentazione e normalizzazione dell'iride. Vediamo nel dettaglio in che cosa consiste.

3.1.1 Preprocessing

Individuazione della ROI

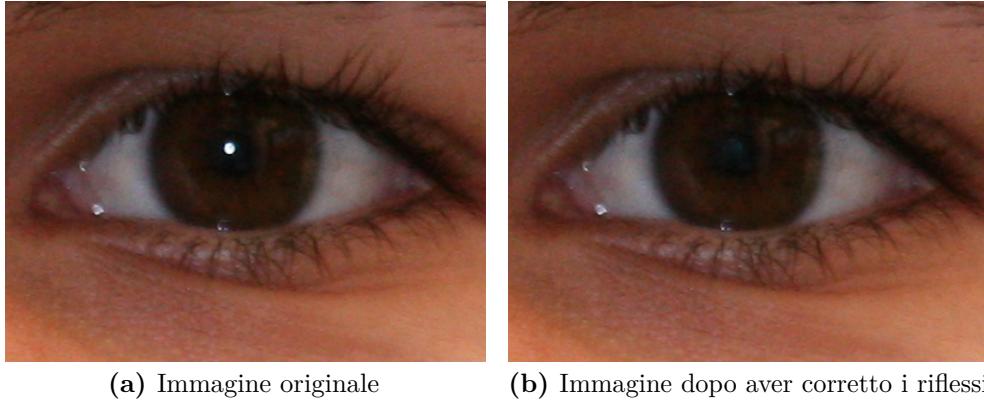
Attraverso un classificatore fornito da OpenCV, addestrato nell'individuazione degli occhi, viene individuata un'area di interesse (o in inglese ROI, Region of Interest) più specifica, nel nostro caso l'area dell'immagine che contiene uno dei due occhi del soggetto.

Questa operazione viene effettuata sia per eliminare eventuali elementi di disturbo che potrebbero portare ad errori di segmentazione, sia per ridurre notevolmente il costo computazionale andando ad escludere numerosi pixel non necessari all'operazione di individuazione dell'iride.

Successivamente verrà preso in esame un metodo per decidere quando è necessario impiegare l'utilizzo di questo classificatore.

Individuazione e correzione dei riflessi

Spesso nelle immagini acquisite sono presenti dei riflessi sull'iride, è necessario quindi individuarli e correggerli. *IS_{IS}* calcola la maschera dei riflessi attraverso il metodo di *adaptiveThresholding* per poi successivamente passarla in input all'algoritmo di *inpainting* di *Telea* [6] in modo tale da correggerli.



(a) Immagine originale

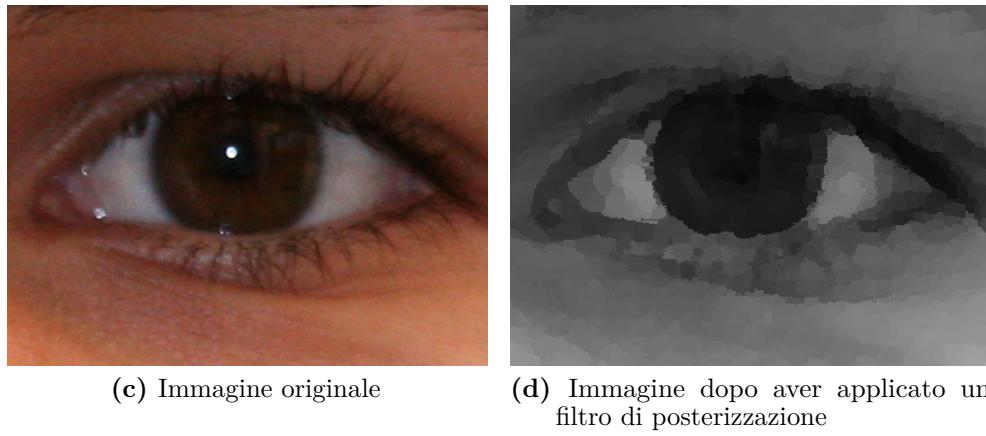
(b) Immagine dopo aver corretto i riflessi

Il calcolo della maschera dei riflessi attraverso l'*adaptiveThresholding* viene effettuato analizzando la componente blu dell'immagine passata in input, nella quale sono più visibili.

Filtro di posterizzazione

Rispetto ad altri metodi, come *Haindl-Krupicka* [8] che applicano semplicemente un filtro Gaussian Blur per sfocare l'immagine, *IS_{IS}* sostituisce questo filtro con uno di posterizzazione così realizzato:

1. L'immagine presa in input viene convertita in scala di grigi.
2. Si fa scorrere una finestra W grande $(k*2+1) \times (k*2+1)$ su tutta l'immagine, e ad ogni scorrimento viene calcolato un istogramma H_W sui pixel interni alla finestra.
3. Il colore più frequente all'interno dell'istogramma viene assegnato al pixel centrale di W .



(c) Immagine originale

(d) Immagine dopo aver applicato un filtro di posterizzazione

Figura 3.1. Sull'immagine a sinistra viene applicato un filtro di posterizzazione con parametro $k = 7$, il risultato è l'immagine a destra

Come si può vedere in figura (3.1), nell'immagine che risulta dopo aver applicato il filtro il contrasto viene aumentato e si riducono i livelli di grigio presenti nell'immagine

originale, questo migliora notevolmente i risultati ottenuti in seguito dal filtro di edge detection.

Successivamente vedremo come l'algoritmo di posterizzazione è stato migliorato in modo tale da garantire una maggiore efficienza.

3.1.2 Segmentazione

L'immagine preprocessata viene data in input al modulo di segmentazione, il quale si occuperà di individuare prima di tutto il cerchio che identifica il limbo e successivamente quello della pupilla. L'approccio utilizzato per il limbo è lo stesso utilizzato per la pupilla.

Viene applicato il filtro di posterizzazione su diverse finestre W_k , $\forall k \in [3, 17]$. Per ogni applicazione del filtro vengono **individuati tutti i possibili cerchi** e per ognuno di questi vengono calcolati i valori di **omogeneità** e **separabilità** del cerchio trovato. La somma di questi due valori costituirà il punteggio che sarà utilizzato come metrica di giudizio: il cerchio con il punteggio più alto verrà considerato il miglior cerchio.

Individuazione dei cerchi

Per individuare i cerchi, subito dopo la posterizzazione viene applicato un filtro di edge detection; precisamente viene utilizzato il metodo Canny [2] attraverso la funzione di libreria *cannyThreshold*.

Sull'immagine risultante, tramite la funzione *findContours*, vengono poi individuati i contorni che a loro volta vengono dati in input all'algoritmo di circle fitting di *Taubin*, riuscendo così a trovare le migliori approssimazioni dei cerchi nel minor tempo possibile.

L'insieme dei cerchi individuati viene poi filtrato attraverso le seguenti condizioni:

- I cerchi candidati ad essere il **limbo** devono avere un raggio r_L , tale che $35 \leq r_L \leq 90$
- I cerchi candidati ad essere la **pupilla** devono rispettare le seguenti condizioni:
 1. Il raggio della pupilla r_P , deve $r_P > 8 \wedge \frac{r_L}{5} < r_P < \frac{r_L}{3}$, dove r_L è il raggio del miglior cerchio individuato come limbo
 2. Il valore di tonalità medio dei pixel all'interno del cerchio deve essere superiore a 30
 3. La distanza tra il centro del cerchio candidato e il centro del cerchio individuato come limbo non deve essere superiore a 15

Per garantire una maggiore efficienza la zona di ricerca della pupilla viene delimitata alla sola area all'interno del cerchio individuato per il limbo.

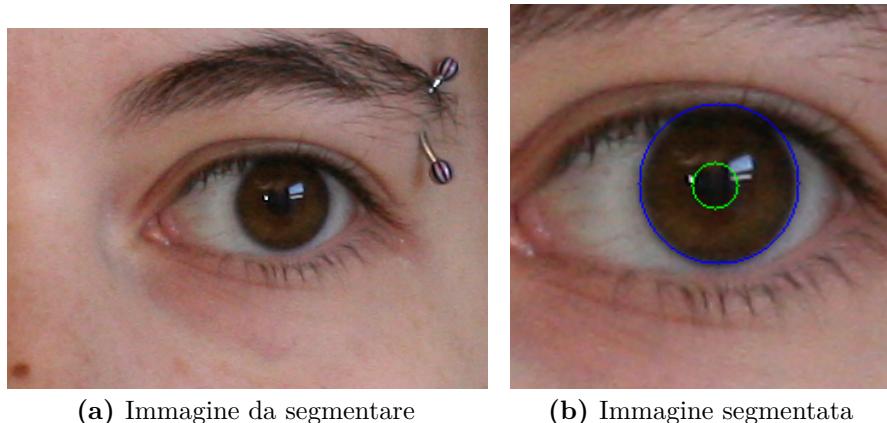


Figura 3.2. Nell’immagine (b) il cerchio blu è quello relativo al limbo, il verde alla pupilla

3.1.3 Normalizzazione

In seguito alla segmentazione si passa alla fase di normalizzazione dell’immagine in quanto nelle fasi successive è necessario poter eseguire delle operazioni senza dover fare nessuna distinzione di posizione, grandezza e orientamento dell’iride.

Il metodo utilizzato, se non altro il più diffuso, è proposto da Daugman in [3] ed è noto come Homogeneous Rubber Sheet Model.

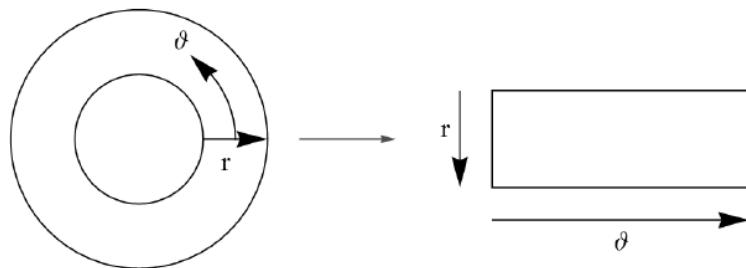


Figura 3.3. Rappresentazione grafica del metodo di normalizzazione utilizzato

Data un’immagine $I(x, y)$ in coordinate cartesiane, la sua trasformazione in un’immagine espressa in coordinate polari $I'(r, \theta)$ avviene nel seguente modo:

$$I(x(r, \theta), y(r, \theta)) \rightarrow I'(r, \theta)$$

dove

$$\begin{aligned} x(r, \theta) &= (1 - r)x_p(\theta) + rx_l(\theta) \\ y(r, \theta) &= (1 - r)y_p(\theta) + ry_l(\theta) \end{aligned}$$

con $(x_p(\theta), y_p(\theta))$ e $(x_l(\theta), y_l(\theta))$ definiti come i punti che si trovano rispettivamente sul contorno della pupilla e del limbo.

Infine viene prodotta la maschera binaria sia dell’immagine espressa in coordinate cartesiane sia dell’immagine dell’iride normalizzata; se un pixel appartiene all’iride sarà bianco, altrimenti sarà nero.

Questa maschera è necessaria per evitare di analizzare in fase di codifica elementi inutili e dannosi al fine di quest’operazione.

3.2 Funzionamento del modulo di codifica e matching

Questo modulo, che prende in input l'immagine normalizzata e la relativa maschera binaria, ha come scopo quello di andare ad estrarre le caratteristiche principali dell'iride che verranno poi utilizzate nella fase di matching per determinare il grado di similarità (o dissimilarità) con le altre codifiche registrate nel sistema.

3.2.1 Codifica

Esistono diversi tipi di codifiche. L'approccio che viene utilizzato per questo lavoro di tirocinio è basato sull'utilizzo e la combinazione di operatori [11], precisamente la combinazione degli operatori **LBP** e **Spatiogram**.

LBP

LBP (Local Binary Pattern) [4] è un operatore utilizzato per l'analisi della tessitura delle immagini.

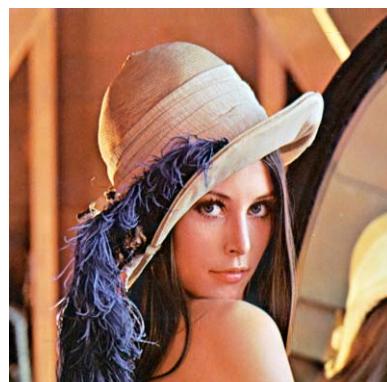
Per applicare questo operatore bisogna scorrere su tutti i pixel dell'immagine e ad ogni iterazione viene creata una finestra 3×3 sulla quale viene posto al centro il pixel preso in esame. Viene poi fissata una soglia s la quale è pari all'intensità del pixel centrale.

Per ogni pixel presente sul bordo p_i viene assegnato un valore attraverso la seguente formula:

$$Value(i) = \begin{cases} 1, & \text{se l'intensità di } p_i > s \\ 0, & \text{altrimenti} \end{cases} \quad (3.1)$$

Successivamente viene scelto un pixel di partenza sul bordo della finestra (quindi escludendo il pixel centrale) e un verso di percorrenza e successivamente viene fatta una linearizzazione ottenendo come risultato un array di 8 elementi (che chiameremo *lbpArray*) che deve essere visto come una rappresentazione binaria di un numero tra 0 e 255. Questo valore viene trasformato in base decimale ed assegnato al pixel preso in esame

$$LBP(x, y) := \sum_{i=0}^7 2^i * lbpArray[i]$$



(a) Immagine sorgente



(b) Immagine dopo aver applicato LBP

Dopo aver ottenuto l'immagine con l'operatore applicato si passa alla codifica: l'immagine è divisa in k fasce orizzontali e per ognuna di queste viene calcolato il relativo istogramma $H_i, \forall i \in [1, k]$.

Alla fine di questa operazione otterremo una codifica così definita:

$$LBP_CODE = \{H_1, \dots, H_k, Mask\}$$

Spatiogram

Uno spatial histogram o anche chiamato **Spatiogram** permette di ottenere non solo informazioni riguardo le occorrenze dei vari colori come farebbe anche un qualsiasi istogramma, ma anche informazioni riguardo la posizione dei pixel che lo compongono.

Per quanto riguarda la codifica di immagini andiamo a considerare uno spatiogram di secondo ordine che possiamo rappresentare con una tripla:

$$h_I^2(b) = \langle n_b, \mu_b, \Sigma_b \rangle$$

dove b indica il bin a cui ci riferiamo, n_b è il numero di occorrenze di un colore appartenente al b -esimo bin, μ_b è il mean vector delle coordinate del pixel e Σ_b è la loro matrice di covarianza.

Per prima cosa viene calcolato l'istogramma dell'immagine (al contrario di LBP l'immagine non viene suddivisa in fasce ma viene elaborata interamente). Viene poi fatto un cambio di coordinate portandole in una scala da -1 a 1 e poi vengono calcolati il mean vector e le matrici di covarianza attraverso le funzioni presenti nell'implementazione precedente [11].

Andiamo poi a definire $SPATIOGRAM_CODE = \{\langle n_b, \mu_b, \Sigma_b \rangle \mid \forall b \in [1, B]\}$

3.2.2 Matching

Per il calcolo della similarità tra due codifiche è stato utilizzato un approccio che prevede la combinazione dei due operatori precedentemente introdotti.

Per fare ciò, innanzitutto abbiamo bisogno di due funzioni che calcolano la similarità tra due codifiche dello stesso tipo così definite:

$$lbpMatch : \{Clbp_1, Clbp_2\} \rightarrow [0, 1]$$

$$spatioMatch : \{Cspatio_1, Cspatio_2\} \rightarrow [0, 1]$$

dove lo 0 del codominio indica che le due codifiche sono identiche mentre l'1 indica che le due codifiche sono completamente diverse.

Per ottenere il grado di similarità finale semplicemente viene fatta una media dei risultati restituiti dalle due funzioni.

Funzione di matching per LBP

Date due codifiche LBP, così definite $C_1 = \{H_1, \dots, H_n, Mask_1\}$, $C_2 = \{K_1, \dots, K_n, Mask_2\}$, il loro grado di similarità è così calcolato:

$$lbpMatch(C_1, C_2) = \frac{1}{n} \sum_{i=1}^n \rho(H_i, K_i) \left(1 - \frac{noise_i}{totpixel}\right)$$

Dove:

- $\rho(H_i, K_i)$ è una funzione (in OpenCV è già implementata) che confronta gli istogrammi tramite il metodo di Bhattacharyya
- $noise_i$ è la media dei pixel di rumore delle maschere $Mask_1$ e $Mask_2$ nell'i-esima fascia

Funzione di matching per Spatiogram

Date due codifiche Spatiogram, così definite $C_1 = \{< n_b, \mu_b, \Sigma_b > \mid \forall b \in [1, B]\}$ e $C_2 = \{< n'_b, \mu'_b, \Sigma'_b > \mid \forall b \in [1, B]\}$, il loro grado di similarità è calcolato con la seguente formula già implementata in [11]:

$$spatioMatch(C_1, C_2) = \sum_{b=1}^B \sqrt{n_b, n'_b} \left[8\pi |\Sigma_b, \Sigma'_b|^{\frac{1}{4}} N(\mu_b; \mu'_b, 2(\Sigma_b + \Sigma'_b)) \right]$$

3.2.3 Performance del riconoscimento tramite operatori

Di seguito vengono riportati i risultati presentati in [11] per quanto riguarda la combinazione di LBP e Spatiogram su immagini segmentate con il metodo proposto da *Haindl-Krupicka* [9].

Modalità di verifica

Abbiamo due tipi di verifiche:

1. **Single-match:** avviene prendendo in considerazione ogni elemento del probe ed effettuando un numero di tentativi di verifica pari al numero di template nella gallery; ogni qualvolta avviene un confronto con un template relativo a se stesso questo verrà considerato genuino, altrimenti sarà considerato impostore.
2. **Multiple-match:** avviene confrontando il soggetto del probe con un gruppo di template appartenenti tutti allo stesso individuo e infine viene riportato solo il valore più basso calcolato. In questo caso per ogni elemento del probe viene effettuato un numero di tentativi pari al numero di soggetti nella gallery, dove risulta genuino in uno solo di essi e impostore in tutti gli altri casi.

Tipo di verifica	Equal Error Rate	Threshold
Single-match	0.3687	0,2174
Multiple-match	0.2189	0.1252

Tabella 3.1. Risultati ottenuti per i test in **modalità di verifica**

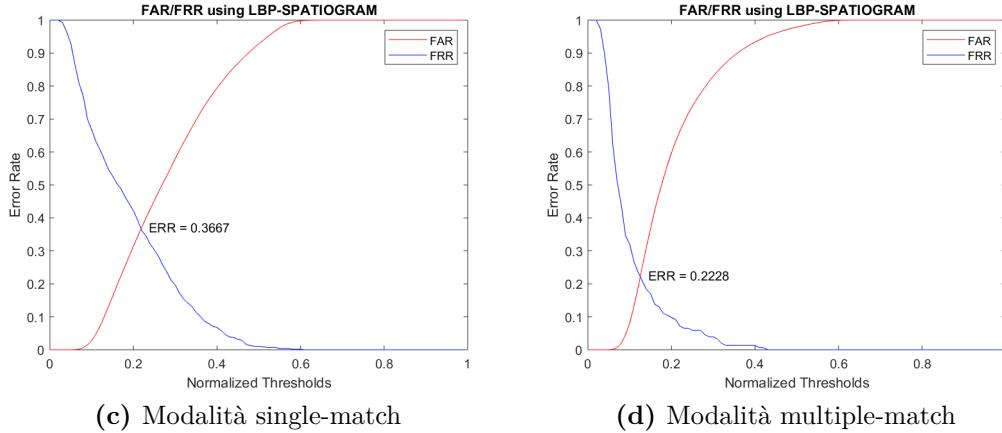


Figura 3.4

Modalità di identificazione

Per i test in **modalità di identificazione** ogni soggetto del probe viene confrontato con tutti i soggetti appartenenti alla gallery ed al termine di quest'operazione viene riportata una lista di tutti i template con cui è avvenuto il confronto, ordinata dal migliore al peggior valore di matching ottenuto.

Alla fine di questi test è stato riportato un **Recognition Rate** (RR) pari a 73; questo valore rappresenta il numero di volte in cui il soggetto del probe preso in input appare primo nella lista ottenuta.

3.3 Risultati dell'integrazione dei due codici

In seguito all'integrazione dei due codici sono stati sviluppati un sistema di enrolling ed un piccolo sistema di identificazione che data un'immagine in input restituisce le 5 immagini più simili registrate nel sistema con il relativo grado di similarità.

La prima cosa che si è notata è la straordinaria velocità acquisita dal programma grazie al semplice porting del codice in un linguaggio come C++.

Durante il lavoro di integrazione sono state notate delle procedure nella precedente implementazione di *ISIS v.2* non ottimizzate, molte volte davvero dispendiose computazionalmente; successivamente vedremo come queste verranno ottimizzate per garantire la massima efficienza dell'elaborazione.

È stato poi possibile notare che il modulo di codifica funziona molto bene, ma solo quando la segmentazione è ottima e se le immagini non presentano troppo rumore, questo però dipende quasi solo esclusivamente dal modulo di segmentazione e dalle immagini acquisite.

Per alcuni problemi di questo modulo sono state trovate delle soluzioni, ma solo parziali, ovvero, non è stato trovato un discriminante che identifichi quando un determinato problema occorre e di conseguenza non è stato possibile implementare un sistema automatico di correzione per quello specifico problema; più avanti verranno comunque presentate e descritte nel dettaglio queste soluzioni. Per altri problemi invece sono stati provati alcuni approcci ma non hanno portato ad ottimi risultati e conseguentemente sono stati scartati.

Uno dei problemi di questo modulo per cui è stata trovata una soluzione almeno parziale riguarda l'uso del classificatore che viene utilizzato per individuare l'area di interesse dell'occhio, infatti se l'occhio occupa già l'intera immagine il detector restituirà dei risultati inaffidabili.

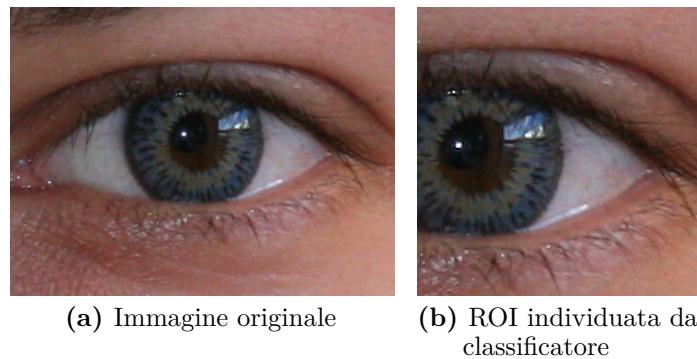


Figura 3.5. Possiamo vedere nell'immagine (b) come una parte dell'iride venga tagliata erroneamente dal classificatore

Su immagini in cui l'occhio è già l'unico elemento dell'immagine il classificatore utilizzato potrebbe non trovare nessun occhio o peggio ancora potrebbe generare un output completamente errato, portando certamente ad una segmentazione errata.

Una buona parte del lavoro di questo tirocinio si basa appunto sull'ideazione e l'implementazione di un sistema che possa permetterci di decidere quando impiegare o meno l'uso di un classificatore di occhi.

Capitolo 4

Miglioramento di *ISIS v.2*

Dati i risultati poco incoraggianti ottenuti dopo aver testato il sistema implementato si è deciso innanzitutto di ottimizzare e migliorare le porzioni di codice in cui l'implementazione già presente era un malus per le prestazioni e in seguito cercare delle soluzioni a problemi presenti all'interno del modulo di segmentazione.

4.1 Nuova implementazione del filtro di posterizzazione

Nella vecchia implementazione del filtro di posterizzazione, ad ogni nuova finestra creata veniva calcolato un nuovo istogramma su tutta la finestra. Così facendo venivano considerati più volte per il conteggio gli stessi pixel probabilmente utilizzati nella finestra precedentemente creata; è da ciò che deriva l'intuizione che c'è alla base del miglioramento dell'algoritmo di applicazione del filtro.

Infatti nella nuova implementazione il filtro viene calcolato nel seguente modo:

Per ogni riga viene generata una nuova finestra, ne viene calcolato l'istogramma e viene assegnato il valore più frequente al pixel centrale della finestra (ovvero il pixel nella prima colonna).

Successivamente viene fatta scorrere la finestra e per ogni riga della finestra vengono:

1. Decrementate dal conteggio dell'istogramma le frequenze dei colori presenti nella prima colonna della vecchia finestra
2. Aumentate sul conteggio dell'istogramma le frequenze dei colori presenti nell'ultima colonna della nuova finestra.

Viene poi calcolato il nuovo colore più frequente e viene assegnato al pixel centrale della finestra.

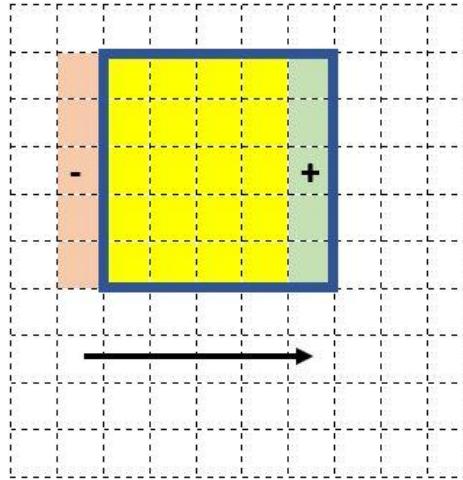


Figura 4.1. Rappresentazione grafica di ciò che avviene ogni volta che la finestra scorre sulle colonne

La vecchia implementazione (3.1.1) aveva una complessità computazionale pari a $O(n^2(k^2 + B))$, dove n^2 rappresenta il numero di pixel presenti nell'immagine, k^2 il numero di pixel di ogni finestra e B il numero di bins che compongono l'istogramma.

Analizziamo ora il costo della nuova implementazione. Abbiamo due possibili scenari:

1. **Siamo sulla prima colonna:** questa situazione capita $O(n)$ volte e quindi $O(n)$ volte dovremmo calcolare l'istogramma per tutta la finestra, quindi la complessità in questo caso è $O(n(k^2 + B))$
2. **Siamo su qualsiasi altra colonna:** questa situazione capita $O(n^2)$ volte, ma in questo caso non dovremmo calcolare l'istogramma su tutta la finestra, ma dovremmo valutare soltanto 2 colonne su $O(k)$ righe, quindi la complessità finale sarà $O(n^2(k + B))$

Per migliorare ulteriormente questa implementazione è stata tolta la B dalla complessità computazionale, andando non più a cercare il colore più frequente all'interno dell'istogramma ogni qualvolta viene fatta scorrere la finestra, bensì ad ogni aggiornamento dell'istogramma teniamo traccia del colore più frequente tenendolo in una variabile di appoggio.

Per completezza di seguito viene riportato l'algoritmo in pseudocodice simil python:

```

def posterization(img, k):
    # inizializza immagine in output
    output = zeros(img.cols, img.rows)
    for y in range(len(img)):
        # crea una finestra con al centro il pixel img[y][0]
        window = new SlidingWindow(img, y, 0, k)
        # colore piu' frequente
        topColor = -∞

        # calcola l'istogramma sulla finestra
        # e restituisce il colore piu' frequente
        histo, topColor = calcHist(img, finestra)

        # setta il pixel con il valore della posterizzazione
        output[y][0] = topColor

        # scorri fra le colonne
        for x in range(len(img[y])):
            # se e' la prima --> skip
            if x == 0: continue
            newWindow = new SlidingWindow(img, y, x, k)
            # scorri solo sulle righe della finestra
            for wy in window:
                # togli dal conteggio le occorrenze dei pixel
                # sulla prima colonna della vecchia finestra
                histo[img[wy][window.startX]] -= 1

                # aggiungi al conteggio le occorrenze dei
                # pixel sull'ultima colonna della nuova finestra
                c_tmp = histo[img[wy][newWindow.endX]]
                histo[c_tmp] += 1

                # tieni traccia del colore piu' frequente
                if histo[topColor] < histo[c_tmp] : topColor = c_tmp

            window = newWindow
            # assegna il colore piu' frequente
            output[y][x] = topColor

    return output

```

4.2 Ottimizzazione ricerca limbo e pupilla

Nell'approccio bruteforce per il rilevamento del limbo e della pupilla, la vecchia implementazione in Java per trovare il miglior limbo o la migliore pupilla prevedeva un ordinamento della lista che li contenesse (in base allo score ottenuto) per poi prendere quello a fine lista, ovvero quello con il punteggio più alto.

Questo aggiungeva una complessità computazionale pari a $O(n^2)$ con n pari al numero dei cerchi trovati ed essendo questi numerosi risulta esserci un costo che può essere evitato e migliorato.

Sulla falsariga dell'implementazione del filtro di posterizzazione, è stato applicato lo stesso concetto utilizzato per la rimozione della B nella complessità computazionale, quindi anziché ordinare ogni volta la lista per poi prendere il migliore, è possibile tenere traccia del miglior cerchio ogni volta che se ne individua uno nuovo, mantenendo informazioni solo sul "migliore" (identificato appunto dallo score), così da ridurre drasticamente e facilmente la complessità ad un semplice $O(1)$, che non è la complessità del codice di individuazione dei cerchi, ma solo della ricerca del migliore tra tutti quelli individuati.

4.3 Soluzioni parziali

Durante la fase di miglioramento del modulo di segmentazione sono state trovate alcune soluzioni per alcuni specifici casi, anche se non è stato possibile individuare un criterio da poter utilizzare per decidere se applicare o meno un'eventuale soluzione per quel particolare problema; queste soluzioni le chiameremo **soluzioni parziali**.

Di seguito ne verranno elencate alcune.

4.3.1 Utilizzo di un filtro di blurring

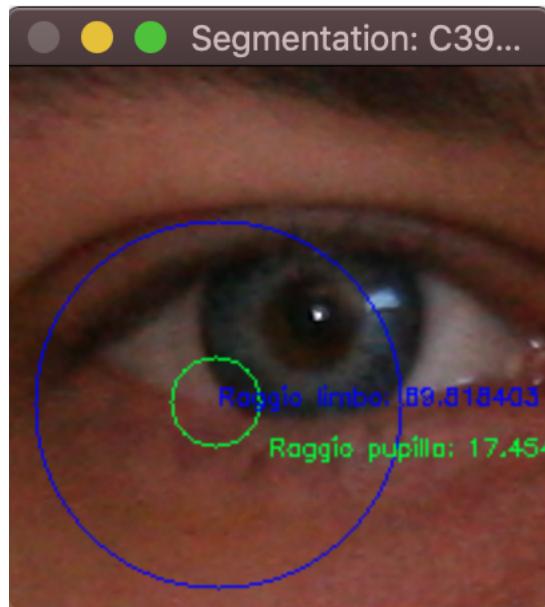


Figura 4.2. Immagine segmentata male

Come si può vedere dalla figura (4.2), quest'immagine ha subito una segmentazione errata, infatti sia il limbo (evidenziato in blu) e conseguentemente la pupilla (in verde) individuati sono posizionati in un punto che non corrisponde all'iride.

Questo è un problema che spesso capita con questo metodo di segmentazione, ma in alcuni di questi casi la segmentazione può essere corretta facilmente applicando il filtro di *medianBlur* con parametro $k = 3$ prima di posterizzare l'immagine.

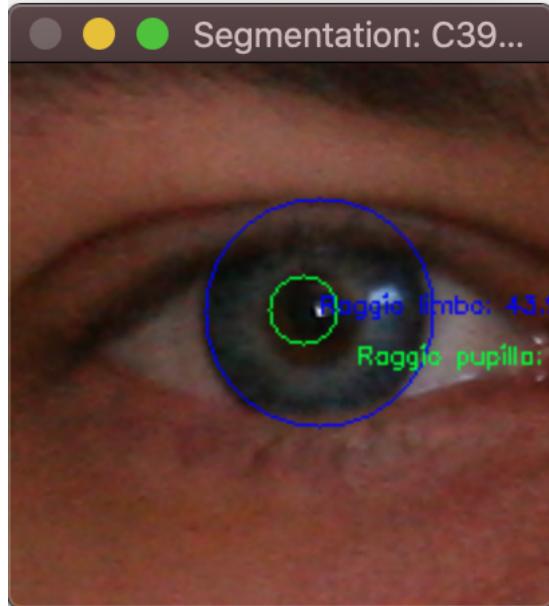


Figura 4.3. Segmentazione con filtro di medianBlur applicato

Infatti come possiamo vedere nella figura 4.3, dopo aver applicato il filtro l'iride viene segmentata meglio, seppur non perfettamente sulla zona della pupilla.

Questa purtroppo è una soluzione parziale, in quanto non si è riusciti a trovare un criterio per decidere quando applicare o meno questa correzione.

4.3.2 Miglioramento individuazione dei riflessi

Nella precedente implementazione di *ISIS v.2* per individuare e creare la maschera dei riflessi è stata sostituito il *simpleThresholding* con l'*adaptiveThresholding* in quanto risultava molto più efficace per l'operazione di filtraggio dei riflessi.

Durante il testing della nuova implementazione questa affermazione viene smentita in quanto alcune immagini che venivano segmentate male, vengono correttamente segmentate tramite l'impiego del *simpleThresholding*.

Purtroppo però anche questa ricade nella categoria delle soluzioni parziali in quanto non è stato possibile trovare un sistema che decidesse in maniera autonoma quando è meglio usare uno e quando l'altro.

Si è quindi pensato che si potessero utilizzare entrambi i metodi di thresholding, per poi andare a scegliere il miglior limbo e la miglior pupilla individuata dai due output ottenuti. Purtroppo però questa soluzione è parecchio dispendiosa e inoltre non sempre funziona. Ciò che possiamo trarre di certo da tutto ciò è che non sempre l'*adaptiveThresholding* è la soluzione migliore.

4.4 Tentativi falliti

Vale comunque la pena citare alcuni tentativi che sono stati effettuati per migliorare il processo di segmentazione in quanto potrebbero essere d'ispirazione per soluzioni migliori, e infatti proprio per questo il codice relativo a questi tentativi è presente nell'implementazione in apposite funzioni ma, ovviamente, non viene utilizzato.

4.4.1 Positioning di un cerchio

Alcune delle immagini utilizzate per i test presentavano, come abbiamo visto anche in figura (4.2) dei cerchi completamente fuori dalla zona di interesse dell'iride.

Durante la fase di debugging di questi problemi è stato possibile notare che solo in pochi casi, molti dei cerchi individuati ad ogni applicazione del filtro di posterizzazione, che poi in seguito sono stati scartati per il basso score raggiunto, erano cerchi che se utilizzati avrebbero individuato quasi perfettamente il limbo o la pupilla. Essendo numerosi i cerchi appena descritti (ricordiamo però solo in pochi casi), si è pensato che per individuare il cerchio migliore si potesse eseguire una media della posizione e del raggio in modo tale da ottenerne una posizione e una grandezza approssimativa.

Per essere più precisi, dato $C = \{ c_i \mid c_i \text{ è individuato all'i-esima applicazione del filtro di posterizzazione} \}$, ed ogni c_i è definito come la tripla $\langle centerX, centerY, radius \rangle$. Calcoliamo quindi c_{avg} , come:

$$c_{avg} = \left\langle \frac{1}{|C|} \sum_{c_i \in C} c_i[centerX], \frac{1}{|C|} \sum_{c_i \in C} c_i[centerY], \frac{1}{|C|} \sum_{c_i \in C} c_i[radius] \right\rangle$$

Purtroppo però questo approccio non è stato utilizzato e si è rivelato un fallimento in quanto anche un solo cerchio molto grande o distante dagli altri avrebbe prodotto un cerchio completamente sbagliato ed inoltre non sono molto frequenti i casi in cui la maggior parte dei cerchi generati sono molto simili tra loro e corretti.

Un altro approccio a questa tecnica potrebbe essere quello di calcolare c_{avg} con una media ponderata (in base agli score ottenuti dai cerchi, ad esempio) e allo stesso tempo tenere traccia del miglior cerchio c_{best} (come per il metodo utilizzato normalmente) e infine calcolare lo score di c_{avg} e restituire il cerchio con il punteggio maggiore.

Capitolo 5

Lavoro sugli istogrammi

Come già accennato nei precedenti capitoli, la seconda parte di questo lavoro di tirocinio si incentra su uno studio per individuare un metodo per decidere quando è necessario utilizzare un classificatore di immagini per individuare l'area di interesse di uno dei due occhi.

La necessità di effettuare questo studio, come già annunciato nel paragrafo 3.1.1, nasce dalla presenza di un problema molto importante quando viene utilizzato un classificatore su immagini che già contengono la sola area di interesse dell'occhio. Il classificatore in questo caso specifico genera un output errato e il risultato che si ottiene è un'area che molto probabilmente non ha niente a che vedere con l'occhio presente nell'immagine, o in altri casi, come in figura (3.5), l'occhio individuato viene tagliato in parte e questo porterebbe sicuramente ad una segmentazione errata dell'occhio, propagando l'errore nei moduli successivi di encoding e matching.

Questi problemi, in *ISIS v.2* [12] venivano in parte risolti nel seguente modo: il crop viene effettuato **se e solo se** viene individuata dal classificatore almeno un'area che potrebbe potenzialmente essere un occhio; tra tutte quelle trovate viene scelta la più grande e su questa vengono effettuati dei controlli sulla grandezza: se è sufficientemente grande allora si può effettuare il crop dell'immagine sull'area individuata.

Purtroppo però questi semplici controlli non bastano nella maggior parte dei casi, in quanto il classificatore potrebbe individuare un'area molto grande, ma comunque non relativa ad uno dei due occhi.

5.1 L'idea

L'idea per l'implementazione di questo modulo si basa sull'intuizione che le immagini, in cui il soggetto principale è già un occhio, avranno una frequenza del colore della pelle molto bassa o avranno una frequenza dei colori dell'occhio molto alta.

Ciò che si intende realizzare è una funzione di controllo che, data un'immagine, calcola la frequenza dei vari colori che la compongono e in base a questi determina se è necessario dare in input l'immagine ad un classificatore di occhi.

La funzione sarà così definita:

$$needClassifier(img) = \begin{cases} true, & \text{se ha bisogno del classificatore} \\ false, & \text{altrimenti} \end{cases} \quad (5.1)$$

È necessario quindi trovare, dopo una fase di analisi, dei colori e delle frequenze che facciano da discriminanti per l'output dell'algoritmo.

L'unico requisito di questa procedura è che debba avere un costo computazionale molto basso, in quanto il costo maggiore di tutto il processo di riconoscimento dell'iride grava già pesantemente sul modulo di segmentazione ed è quindi necessario non rallentarlo ulteriormente perché è utilizzato sia nella fase di enrolling che di riconoscimento. Possono quindi essere effettuati molteplici controlli, l'importante è che questi siano effettuati in modo tale da essere molto veloci.

Alla base della soluzione proposta c'è il calcolo dell'istogramma dell'immagine il quale viene effettuato velocemente e ci fornisce tutte le informazioni necessarie riguardo le frequenze dei colori presenti nell'immagine. Per effettuare quest'operazione è sufficiente la funzione *calcHist* la quale attraverso i vari parametri ci consente di poter personalizzare anche l'istogramma che stiamo calcolando.

5.2 Primo approccio: RGB

Un primo approccio potrebbe essere quello di calcolare l'istogramma sullo spettro di colori **RGB** (Red, Green, Blue) per vedere se esiste un colore che è molto più frequente rispetto agli altri.

Ogni pixel in questo spettro di colori è rappresentato da una tripla:

$$\langle r, g, b \rangle \text{ con } r, g, b \in [0, 255]$$

Quindi potenzialmente potremmo avere un istogramma per 256^3 colori differenti. Possiamo facilmente intuire che due o più colori molto simili tra loro verrebbero sicuramente conteggiati come colori differenti, rendendo questo approccio inutilizzabile.

5.3 Passaggio allo spettro di colori HSV

Nasce quindi la necessità di passare all'utilizzo di un nuovo spettro di colori che ci consenta di poter rilevare con facilità il **tipo di colore** presente su un determinato pixel.

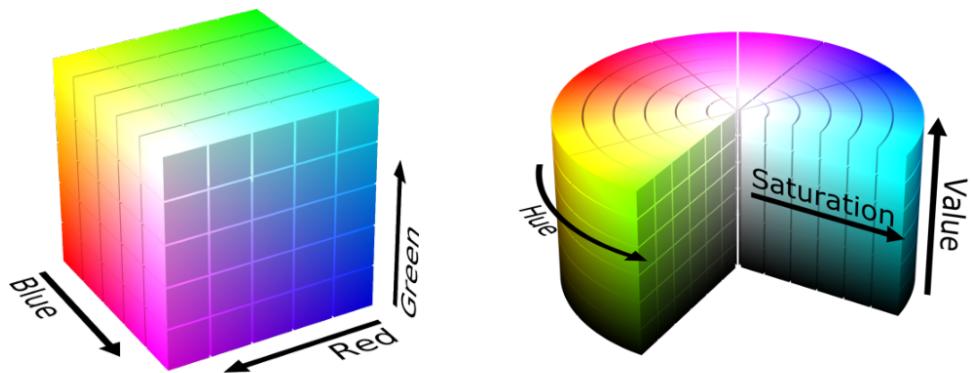


Figura 5.1. Rappresentazione grafica degli spettri RGB e HSV

Lo spettro che è stato preso in considerazione da qui in avanti si chiama **HSV** (Hue Saturation Value) e il canale su cui ci concentreremo è l'Hue attraverso il quale è possibile identificare facilmente la tonalità di colore di un pixel.

Ogni pixel rappresentato in questo spettro di colori, come già detto è rappresentato dai valori di Hue, Saturation e Value.

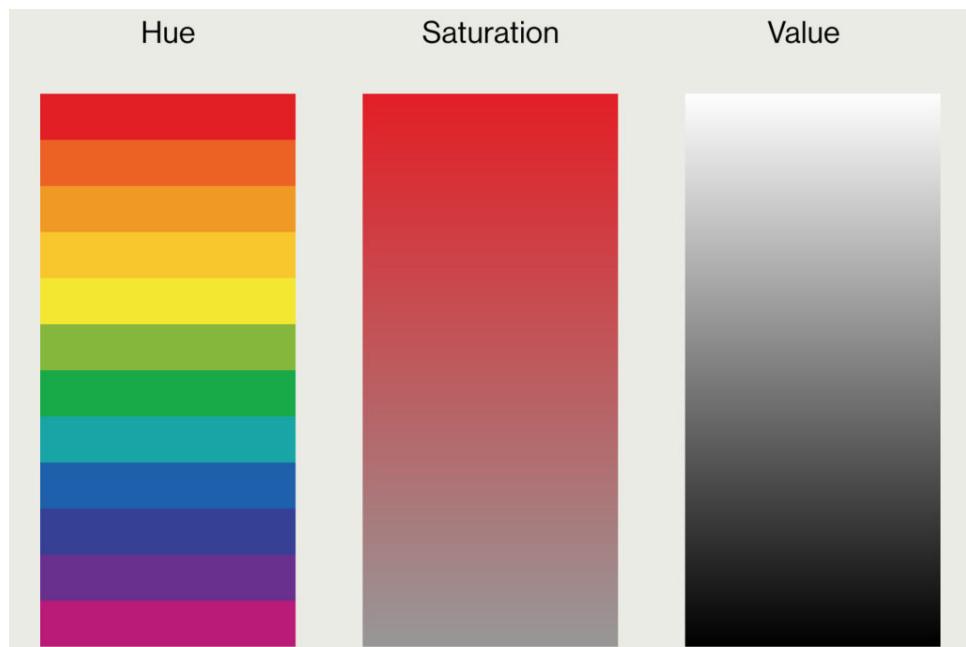


Figura 5.2. Rappresentazione grafica dello spettro di colori HSV

Soltanamente questi tre canali vengono rappresentati da triple del tipo:

$$\langle hue, sat, val \rangle$$

dove:

- $hue \in [0, 360)$ esprime in gradi la tonalità del colore.

- $sat \in [0, 100]$ esprime in percentuale quanto è saturo il colore.
- $val \in [0, 100]$ esprime in percentuale quanto è luminoso il colore.

Per quanto riguarda OpenCV invece i valori vengono mappati differentemente:

- $hue \in [0, 180]$
- $sat \in [0, 255]$
- $val \in [0, 255]$

In OpenCV è già implementata una funzione per la conversione dei colori da RGB a HSV, ovvero la funzione *cvtColor*.

5.3.1 Quantizzazione

Per migliorare i risultati dell'istogramma si è deciso di quantizzare il valore di tonalità (*hue*) in 6 fasce così da mapparlo in 6 possibili tonalità di colore.

Possiamo definire una funzione di mapping *hsv2Bin(hue)* la quale, dato un valore di *hue*, ci restituisce il bin di appartenenza per quella determinata tonalità. Questa funzione sarà molto utile in seguito e sarà così definita:

$$hsv2Bin(hue) = \begin{cases} 0, & \text{se } hue \in [0, 30) \\ 1, & \text{se } hue \in [30, 60) \\ 2, & \text{se } hue \in [60, 90) \\ 3, & \text{se } hue \in [90, 120) \\ 4, & \text{se } hue \in [120, 150) \\ 5, & \text{se } hue \in [150, 180) \end{cases} \quad (5.2)$$

Con le rispettive tonalità:

- 0 corrisponde alla tonalità dell'arancione
- 1 corrisponde alla tonalità del giallo
- 2 corrisponde alla tonalità del verde
- 3 corrisponde alla tonalità del blu
- 4 corrisponde alla tonalità del viola
- 5 corrisponde alla tonalità del rosso

Quantizzando in soli 6 bin, i colori con tonalità simili parteciperanno al conteggio della stessa tonalità di colore e sarà più facile verificare eventuali picchi appartenenti ad una determinata fascia.

5.3.2 Distribuzione delle fasce

Per capire meglio dove fossero distribuite queste fasce di colori sull'immagine si è deciso di definire una funzione molto semplice, ma molto utile in fase di debugging e analisi.

Questa funzione prende in input un'immagine sullo spettro HSV e restituisce un'altra immagine che utilizza lo spettro di colori RGB. Tale funzione è così definita:

$$\text{binsDistribution} : \text{ImgHSV} \rightarrow \text{ImgRGB}$$

Riportiamo brevemente lo pseudocodice della procedura:

```
def binsDistribution (imgHSV):
    # inizializzo la nuova immagine da restituire
    w, h = imgHSV.size ()
    imgRGB = Mat(w, h, 'RGB')

    # scorro sulle due immagini
    for y in range(h):
        for x in range(w):
            # Prendo il bin in corrispondenza di quel pixel e il
            # colore associato e lo assegno all'immagine da restituire
            imgRGB[y, x] = colorFromBin (hsv2Bin (imgHSV[y, x].h))

    return imgRGB
```

5.3.3 Primi risultati

Dopo le premesse fatte nelle sezioni precedenti si è passati al calcolo degli istogrammi prendendo in considerazione poche immagini campione a distanze di acquisizione diverse.

Facendo un plot degli istogrammi calcolati sulle immagini acquisite a breve distanza dall'occhio (esempio in figura 5.3) la cosa che subito si può notare è che il picco massimo in ogni immagine è sulla prima fascia, ovvero quella dei colori con tonalità arancione.

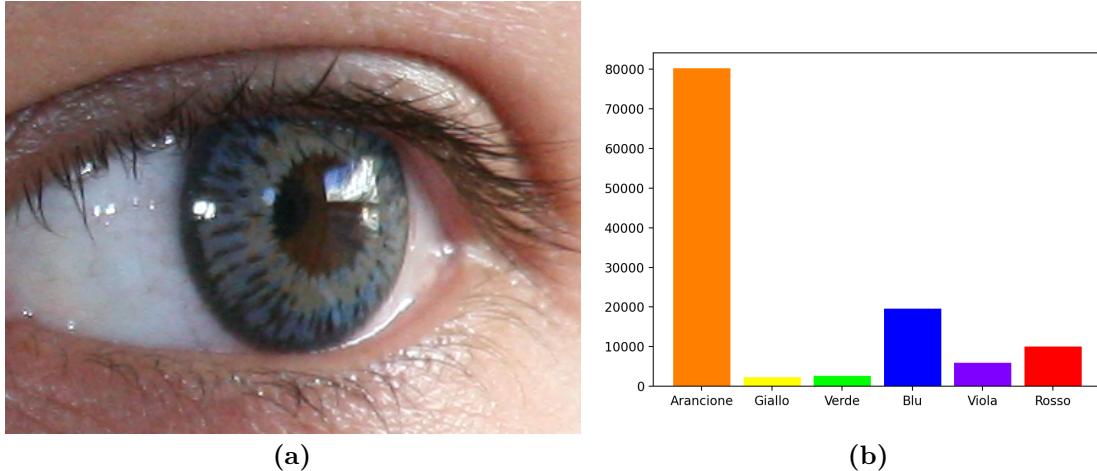


Figura 5.3. Istogramma calcolato per l'immagine (a)

Ovviamente era stato previsto che la fascia in cui ricadessero i pixel della pelle fosse quella dominante, ma non ci si aspettava che lo fosse così tanto rispetto agli altri colori che sono quasi nulli a differenza del blu e del rosso.

Applicando poi la funzione *binsDistribution* all'immagine (a) in figura (5.3) otteniamo in output la seguente immagine:

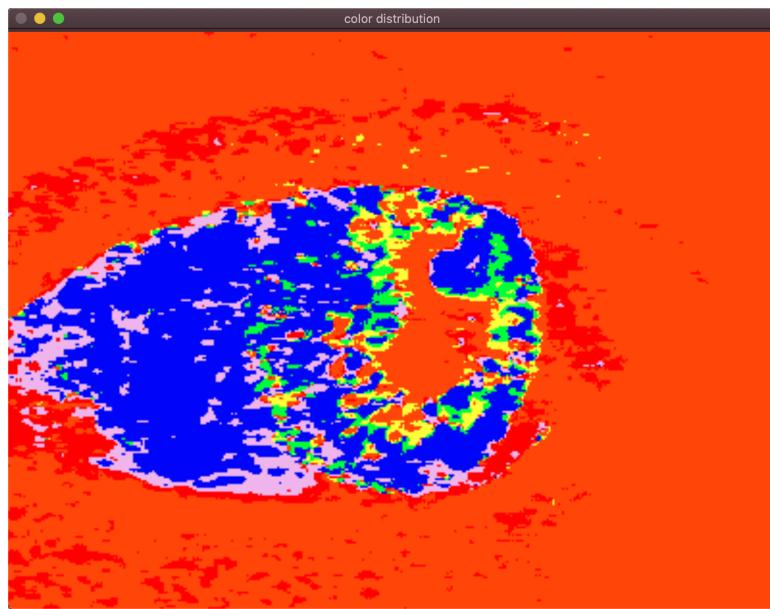


Figura 5.4. Immagine in output dalla funzione *binsDistribution*

Si può chiaramente vedere che le due fasce principali sono l'**arancione** per quanto riguarda la pelle e il **blu** per quanto riguarda la sclera.

Si è poi deciso di applicare lo stesso procedimento ad immagini in cui la distanza di acquisizione rispetto l'occhio fosse maggiore di quella presente nelle immagini precedentemente utilizzate per i test.

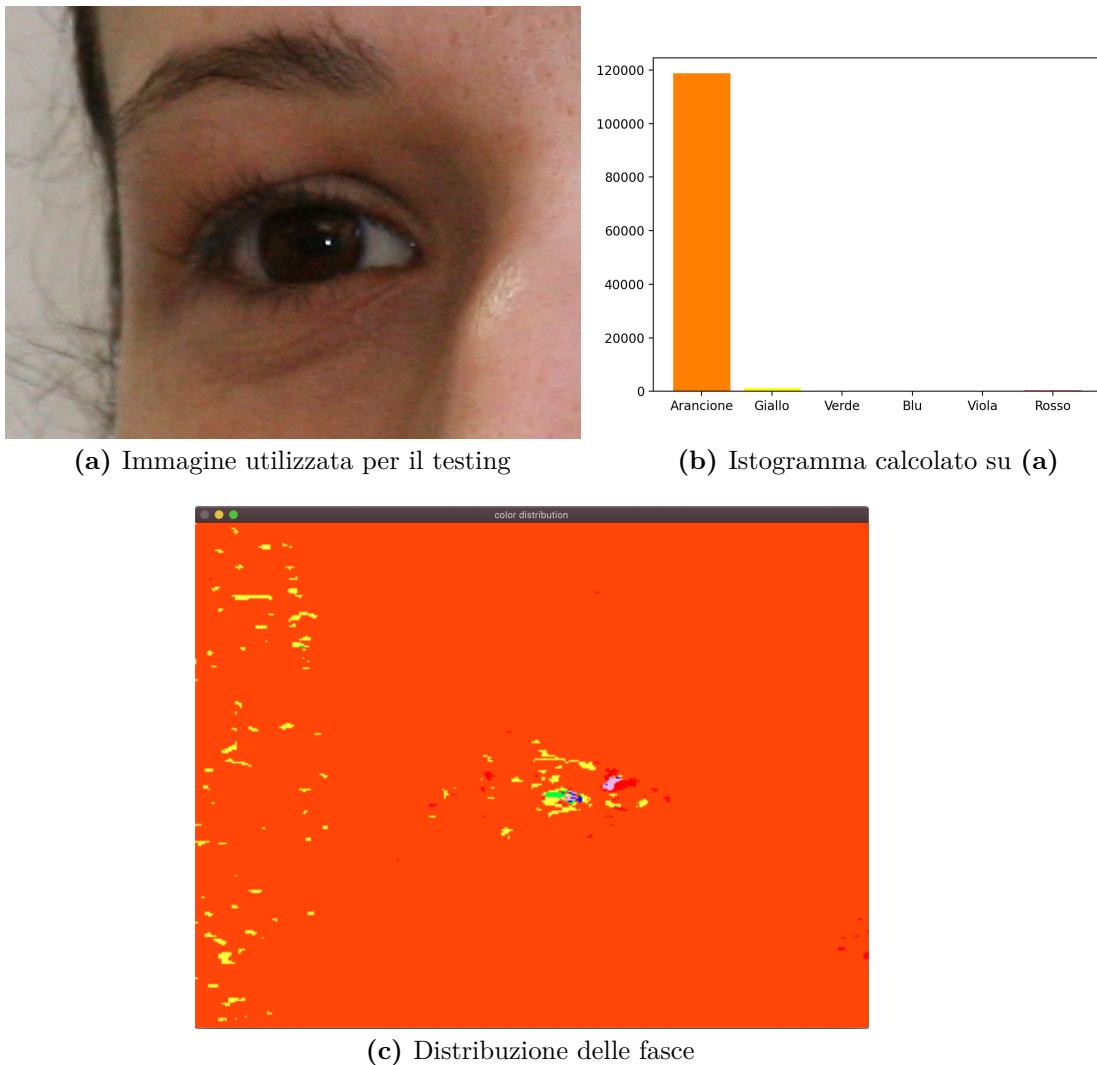


Figura 5.5

Gli istogrammi e le immagini delle distribuzioni delle fasce (esempio in figura 5.5) mostrano come l'unico picco presente sia relativo alla tonalità dell'arancione. Questo inizialmente ha portato a pensare che il semplice valore dell'arancione potesse bastare come discriminante per decidere qualora fosse necessario impiegare l'uso del classificatore.

5.4 Introduzione del dataset Ubiris v2

Dopo aver testato il software su pochi campioni e dopo aver tratto le conclusioni sopra citate è sorta la necessità di dover esaminare più immagini con soggetti, inquadrature e luci differenti. Fortunatamente nel campo del riconoscimento dell'iride sono presenti dei dataset da poter utilizzare in fase di testing o di analisi.

Quello che viene presentato e utilizzato in questo lavoro di tirocinio è chiamato **Ubiris** [7] (versione 2).

Questo dataset contiene immagini riguardanti circa 500 occhi per ognuno dei quali sono state catturate tra le 15 e le 30 foto, contando un totale di 11101 immagini.

La peculiarità di questo dataset riguarda le acquisizioni, le quali:

- vengono effettuate a più distanze
- vengono eseguite in differenti sessioni, una con condizioni controllate l'altra con condizioni non controllate
- vengono eseguite con angolazioni differenti dello sguardo

Per quanto riguarda le caratteristiche cromatiche delle iridi acquisite, il 18.3% presentano una pigmentazione chiara, il 39.1% una pigmentazione scura e il restante 42.6% una pigmentazione intermedia.

Le immagini del dataset hanno una **risoluzione bassa**, pari a 400x300, sono in formato ".tiff" e il dispositivo di acquisizione è una "Canon EOS 5D".

Ogni immagine è composta da una **nomenclatura particolare**:

C<ID_Occhio>_S<ID_Sessione>_I<ID_Img>.tiff

Dove:

- **ID_Occhio** è l'id dell'occhio: se il numero è pari significa che stiamo considerando l'occhio sinistro, altrimenti stiamo considerando l'occhio destro.
Inoltre dati due $ID_Occhio = 2k$ e $2k + 1$, questi saranno appartenenti allo stesso soggetto.
- **ID_Sessione** è l'id della sessione: se questo valore è pari a 1 allora l'immagine è stata acquisita in una sessione controllata, altrimenti se pari a 2 significa che l'acquisizione è avvenuta in una situazione non controllata.
- **ID_Img** è l'id dell'immagine: questo numero ci fornisce informazioni riguardo la direzione dello sguardo e della distanza di acquisizione (qui chiamata d).

Infatti per la distanza:

- $ID_Img \in [1, 3] \rightarrow d = 8m$
- $ID_Img \in [4, 6] \rightarrow d = 7m$
- $ID_Img \in [7, 9] \rightarrow d = 6m$
- $ID_Img \in [10, 12] \rightarrow d = 5m$
- $ID_Img \in [13, 15] \rightarrow d = 4m$

e per lo sguardo:

- $ID_Img \in \{1, 4, 7, 10, 13\} \rightarrow$ sguardo frontale
- $ID_Img \in \{2, 5, 8, 11, 14\} \rightarrow$ sguardo verso destra
- $ID_Img \in \{3, 6, 9, 12, 15\} \rightarrow$ sguardo verso sinistra

La scelta di utilizzare questo dataset rispetto ad un altro (come Utiris ad esempio) sta proprio nell'informazione fornita dall'Id_img riguardo la distanza di acquisizione.

Questa infatti ci consente di poter dividere con estrema facilità il dataset in due insiemi:

1. I_{Far} : ovvero le immagini con una grande distanza di acquisizione e che quindi hanno bisogno del classificatore.
2. I_{Near} : ovvero le immagini con una breve distanza di acquisizione e che quindi non hanno bisogno del classificatore.

Dato il dataset D , definito come l'insieme delle immagini presenti in Ubiris, abbiamo quindi i due rispettivi insiemi così definiti:

1. $I_{Far} = \{ img \mid ID_Img(img) \in [1, 9] \wedge img \in D \}$
2. $I_{Near} = D - I_{far}$

5.5 Analisi massiva

Una volta preparato il dataset, è stato ideato uno script in python che ha il principale compito di raccogliere i dati di ogni istogramma e di inserirli all'interno di un database e all'interno di un file in formato CSV. Una volta ottenuti i dati è stata effettuata un'attenta analisi su ogni fascia.

5.5.1 Fascia del giallo e del verde

Dai dati raccolti da questa prima analisi massiva è emerso che queste due fasce soltanto in 5 immagini risultano essere dominanti. Dato il numero molto basso di immagini presenti in questa categoria si è deciso di verificarle manualmente per capire dove e come fossero distribuiti i colori di queste due fasce.

Da queste verifiche è stato possibile determinare che le immagini che fanno parte di questa categoria o sono immagini scattate su soggetti con una carnagione molto scura o sono immagini scattate con una luminosità molto bassa e che presentano un eccessivo rumore.

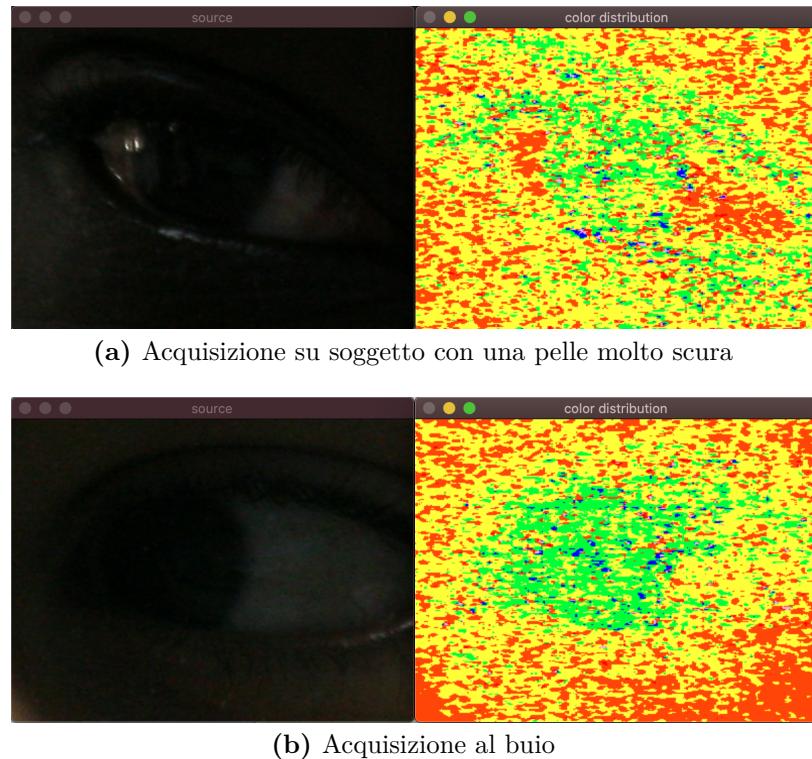


Figura 5.6. Distribuzioni delle tonalità di colore calcolate a partire dalle immagini a sinistra

In seguito si è passati ad esaminare immagini dove queste due tonalità hanno un’alta frequenza (anche senza essere prevalenti nell’immagine), confermando che alte frequenze di verde e di giallo sono dovute, con alte probabilità, ad un forte rumore presente nelle immagini scattate al buio. Questa informazione ottenuta potrebbe essere utilizzata per scartare sin dal principio immagini con forte rumore, in quanto potrebbero rendere l’estrazione delle feature più complessa.

5.5.2 Fascia del blu e del viola

Effettuando le stesse procedure di analisi svolte per il giallo e per il verde, si è potuto notare come il viola non sia in nessuna immagine il colore dominante, mentre il blu lo è in sole due immagini (riguardanti lo stesso soggetto).

Esaminandole si è notato un forte riflesso sugli occhiali del soggetto preso in esame; questo ha portato a pensare inizialmente ad una maggiore distribuzione del blu proprio nel punto di questo riflesso.

L’immagine della distribuzione delle fasce (figura 5.7), smentisce in parte questa affermazione perché evidenzia come la montatura degli occhiali partecipi, in buona parte, al conteggio della frequenza della fascia blu nell’istogramma.

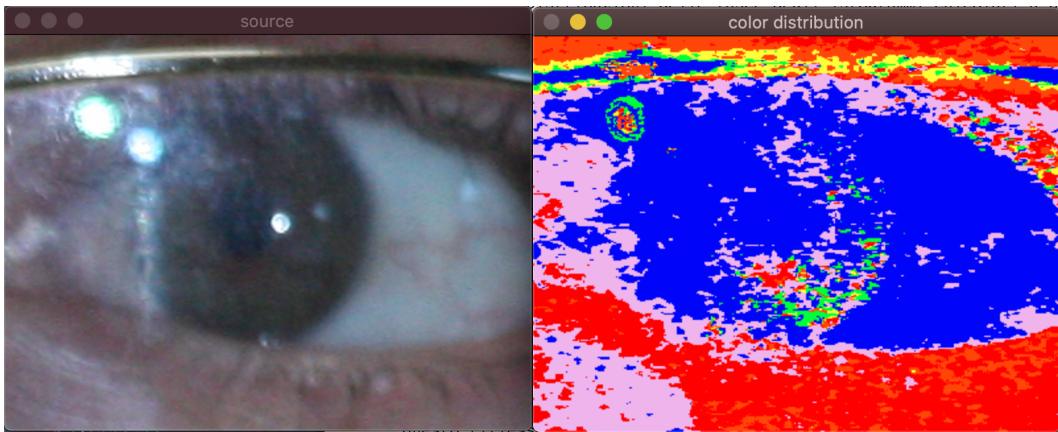


Figura 5.7. A sinistra una delle due immagini in cui il blu è il colore dominante, a destra la distribuzione delle fasce di colori

Analizzando invece le immagini in cui il blu non è la tonalità che prevale, ma sono comunque presenti dei picchi, si è notato grazie alla nomenclatura delle immagini che la maggior parte di queste, avendo un ID_Img molto alto, facessero parte dell'insieme I_{Near} , ovvero le immagini che non hanno bisogno del classificatore. Proprio per questo il blu è una delle due fasce che è stata presa in considerazione come discriminante.

Per quanto riguarda il viola, invece, si è notato come questo sia spesso distribuito sui bordi della sclera, mentre altre volte è distribuito sui riflessi presenti sulla pelle, se questa risulta essere molto chiara.

5.5.3 Fascia dell'arancione e del rosso

Anche senza controllare ulteriormente i dati, già dai risultati delle precedenti fasce, si può evincere che queste due tonalità nella maggioranza dei casi sono le più dominanti in quanto più vicine alla tonalità della pelle. Proprio per questo motivo non è stato possibile applicare lo stesso procedimento di analisi svolto per le altre fasce.

Ciò che si è potuto notare senza analizzare nessuna immagine, semplicemente osservando i dati, è che con frequenze elevate di arancione o di rosso le immagini tendono ad avere degli ID_Img molto bassi, ciò implica che facciano parte dell'insieme I_{Far} e che quindi hanno bisogno del classificatore. Si è subito quindi passati ad analizzare le immagini che invece appartenessero all'insieme I_{Near} e una frequenza di arancione elevata per capire quale fosse il motivo per cui non venissero rilevati altri colori.

Dopo un'attenta analisi si è capito che immagini catturate con una **luce molto calda** tendono facilmente ad estremizzare il valore di Hue di ogni pixel verso lo 0 o verso il suo massimo valore (esempio in figura 5.8).

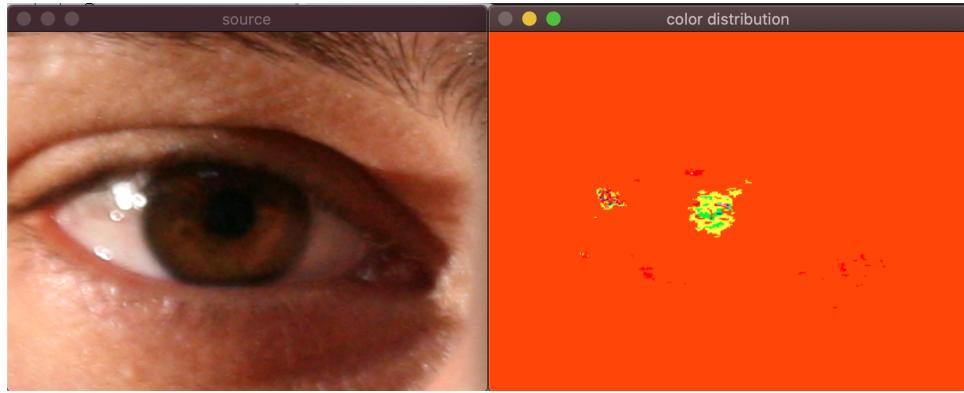


Figura 5.8. Acquisizione con una luce calda e la sua distribuzione

Altri due fenomeni che tendono ad estremizzare questo valore sono l' **arrossamento della sclera** e il **riconfiamento dei capillari** posti tra la sclera e la congiuntiva.

5.5.4 Conclusioni della prima analisi massiva

Dopo questa prima fase di analisi sia la tonalità dell'arancione che del blu sono state prese in considerazione come possibili candidate ad essere valori discriminanti per l'output della procedura che si sta progettando.

Successivamente invece è stato deciso di utilizzare soltanto il blu in quanto è l'unico colore che appare quasi e solo esclusivamente nella zona che riguarda l'occhio, precisamente la sclera, e perché è possibile utilizzarlo su immagini con una scarsa illuminazione e indipendentemente dal colore della pelle del soggetto preso in esame, dato che invece l'arancione in questi casi viene sostituito dal verde e dal giallo. Si è deciso di scartare l'arancione anche perché sono numerose le immagini in I_{Near} nelle quali l'arancione è il colore dominante.

5.6 Individuazione del discriminante

Una volta decisa la tonalità che meglio si presta per la realizzazione della funzione *needClassifier* è stato necessario individuare una soglia t di frequenza del blu che determini il risultato dell'algoritmo che si sta progettando.

È stata posta quindi la funzione *needClassifier* come segue:

$$\text{needClassifier}(\text{img}) = \begin{cases} \text{false}, & \text{se } \text{freqBlu} \geq t \\ \text{true}, & \text{altrimenti} \end{cases} \quad (5.3)$$

Con $\text{freqBlu} = \frac{\text{pixelBlu}}{\text{totPixel}}$, dove:

- pixelBlu è il valore dell'istogramma in corrispondenza del bin del blu.
- totPixel è il numero totale di pixel nell'immagine.

Una volta ottenuta la funzione *needClassifier* resta da trovare un valore di t che ci consenta di poter ottenere i risultati che più fanno al caso nostro.

Per aiutarci in questo processo i due possibili output della funzione *needClassifier* sono stati divisi in quattro categorie:

- **Genuine Acceptance:** $img \in I_{Far} \wedge freq_{Blu} < t$
- **False Acceptance:** $img \in I_{Near} \wedge freq_{Blu} < t$
- **Genuine Rejection:** $img \in I_{Near} \wedge freq_{Blu} \geq t$
- **False Rejection:** $img \in I_{Far} \wedge freq_{Blu} \geq t$

Nella prima fase di ricerca si pensava che una buona soglia di t potesse essere quella in corrispondenza dell'Equal Error Rate (ERR), ovvero il punto in cui t fosse tale che $FAR(t) = FRR(t)$ (rispettivamente **False Acceptance Rate** e **False Rejection Rate**). Purtroppo questo avrebbe portato la funzione a un elevato tasso di errore, in quanto solo attraverso la fascia del blu non è possibile determinare se effettivamente un'immagine ha bisogno del classificatore o meno, quindi si è deciso di passare ad una fase di riprogettazione della funzione.

5.7 Riprogettazione di *needClassifier*

A differenza della precedente progettazione si è deciso di effettuare controlli numerosi e a cascata, in modo tale da poter scartare pian piano le diverse immagini che non necessitano di un classificatore.

La nuova struttura della funzione è del tipo:

```
def needClassifier (img):
    C1 = calcC1()
    if C1:
        return false

    ...
    Ci = calcCi()
    if Ci:
        return false

    ...
    Cn = calcCn()
    return !Cn
```

dove ogni C_i è un controllo che può essere inserito all'interno della funzione. Come si può notare quindi ad ogni controllo verrà deciso se NON passare l'immagine al classificatore o se sottoporla ad ulteriori controlli, fino ad arrivare all'ultimo controllo C_n che darà il giudizio finale.

L'implementazione di controlli a cascata, porta un grande vantaggio per quanto riguarda i casi in cui si è più incerti. Questo perché utilizzando per ogni controllo una soglia che riesca a garantire un basso False Rejection Rate, se su un controllo si è incerti, si può passare l'immagine al prossimo controllo.

5.7.1 Definizione e testing del primo controllo

È stato definito quindi il primo controllo C_1 con la seguente condizione:

$$C_1 = freq_{Blu} \geq t \quad (5.4)$$

con una soglia $t = 0.03$.

I risultati ottenuti con l'utilizzo di questo controllo con questo preciso valore di t sono ottimi:

1. Per quanto riguarda I_{Near} :
 - (a) 2289 immagini su 4460 (51%) non vengono passate al classificatore
 - (b) 2171 immagini su 4460 (49%) superano il primo controllo e vengono passate a quello successivo
2. Per quanto riguarda I_{Far} :
 - (a) 43 immagini su 6641 (0.6%) erroneamente non vengono passate al classificatore
 - (b) 6598 immagini su 6641 (99.4%) superano il primo controllo e vengono passate a quello successivo

Da questi risultati possiamo dedurre che per questo primo controllo abbiamo un errore soltanto nel caso **2a** in quanto sole 43 immagini che hanno bisogno di un classificatore non vengono effettivamente passate a quest'ultimo.

Il caso **1b** non viene considerato come un vero e proprio errore, in quanto queste 2171 immagini potrebbero essere scartate ai prossimi controlli a cui si devono sottoporre.

5.7.2 Migliorie apportate a C_1

Una volta ottenuti i risultati da C_1 , si è pensato che questo controllo avesse un margine di miglioramento. Infatti, nel momento in cui avviene il conteggio dei pixel con tonalità blu, molti di questi sarebbe meglio non considerarli. Un esempio è quello già illustrato nella figura (5.7), dove alcuni dei pixel che venivano considerati per la fascia del blu in realtà non facevano parte della sclera, bensì erano situati o sulla montatura degli occhiali o sul riflesso della lente degli stessi.

La soluzione che viene proposta si basa sulla creazione e l'utilizzo di diverse maschere che unite tra loro individuano i punti che, nel momento in cui viene calcolato l'istogramma, la funzione *calcHist* deve ignorare (se il pixel è spento questo verrà ignorato).

Formalmente, avremo n maschere $\{M_1, \dots, M_n\}$ ognuna delle quali ha come soli pixel "accesi" quelli che si vogliono escludere e una maschera B la quale evidenzia i pixel che hanno tonalità blu. Ciò che vogliamo ottenere è una maschera O , da passare alla funzione *calcHist*, così che sappia quali pixel escludere dal calcolo dell'istogramma.

O è così calcolata:

$$O = \neg((M_1 \mid \dots \mid M_n) \& B)$$

Viene effettuato un OR bitwise tra le n maschere e successivamente la maschera che si ottiene viene messa in AND bitwise con la maschera B , in modo tale da evidenziare solo i pixel che fanno parte della tonalità blu. Infine viene fatta la negazione della maschera in quanto *calcHist* ignora i pixel spenti.

Di seguito verranno riportate alcune delle maschere che sono state implementate.

Maschera riflessi

L'utilizzo di questa maschera risulta molto utile nei casi in cui sono presenti dei piccoli riflessi all'interno dell'iride o ad esempio sugli occhiali dei soggetti presenti nell'immagine, infatti alcuni dei pixel relativi a queste zone in fase di creazione dell'istogramma partecipano al conteggio della tonalità del blu, ma a noi interessa limitare il conteggio alla sola zona riguardante la sclera.

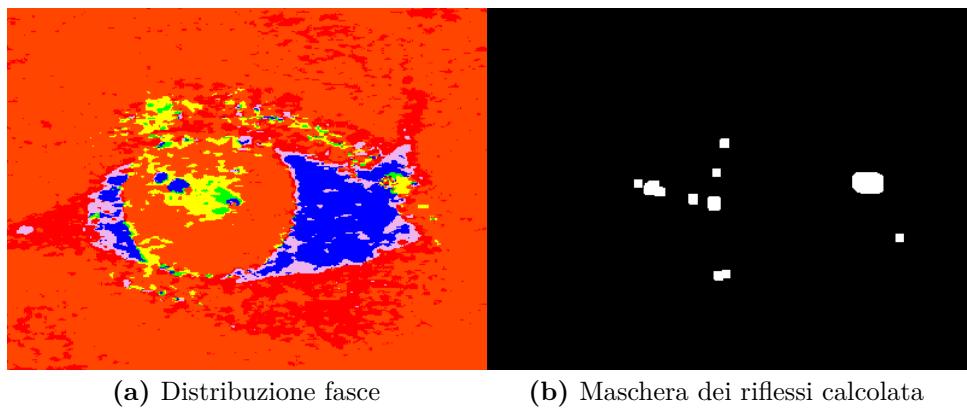


Figura 5.9

La creazione di questa maschera è molto semplice in quanto viene utilizzato l'*adaptiveThresholding* (o anche il *simpleThresholding*), che è la stessa tecnica descritta nel paragrafo (3.1.1) usata per individuare la posizione dei riflessi per poi effettuarne il filtraggio.

Maschera pixel scuri

Come possiamo vedere anche dalle immagini in figura (5.10), in alcuni casi ci sono dei pixel che hanno tonalità blu, ma non hanno niente a che vedere con la sclera.

Dato che il colore di quest'ultima tende al bianco, si è pensato che si potesse creare una maschera che andasse ad evidenziare tutti i pixel scuri presenti nell'immagine originale perché sicuramente non faranno parte della sclera.

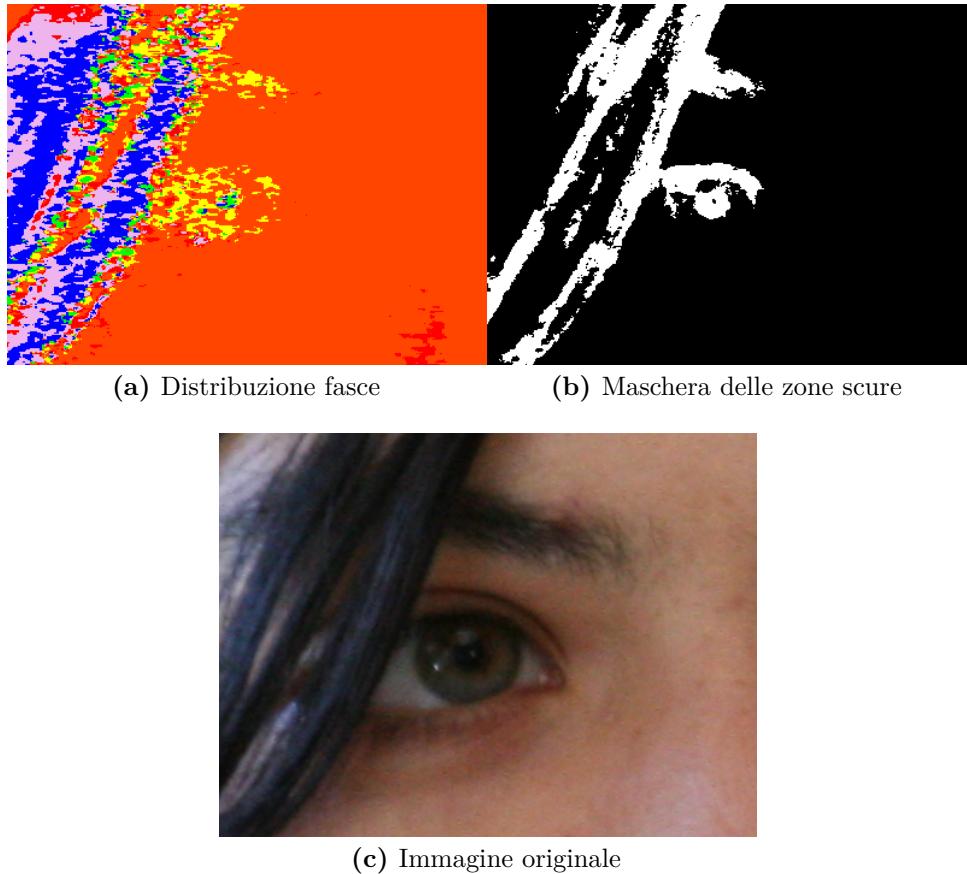


Figura 5.10

L'operazione di creazione di una maschera del genere è molto semplice grazie alla funzione *inRange* presente in OpenCV.

Questa funzione, data un'immagine e due triple HSV l e r , che identificano rispettivamente il limite inferiore e il limite superiore del range di colori che stiamo cercando, restituisce una maschera in cui i pixel accesi hanno un valore HSV all'interno del range.

Il range utilizzato attualmente per individuare la maschera dei pixel scuri è dato da:

- $l = \langle 0, 0, 0 \rangle$
- $r = \langle 180, 255, 30 \rangle$

Come si può notare infatti per i canali di *hue* e *saturation* il range comprende tutti i possibili valori, mentre per quanto riguarda il *value* andiamo a prendere soltanto quelli nell'intervallo $[0, 30]$ in quanto valori maggiori di 30 porterebbero ad un colore molto "luminoso".

5.7.3 Introduzione del secondo controllo

Dalla prima fase di analisi si può dedurre che utilizzare il blu per determinare l'esito di *needClassifier*, funziona molto bene, ma non è sufficiente in quanto nelle immagini con luce calda o con sclere arrossate (come visto in figura 5.8) purtroppo questo metodo non funziona, è stato quindi introdotto ed implementato un nuovo controllo C_2 .

Questo nuovo controllo, come C_1 , si basa sempre sull'individuazione dei pixel della sclera, ma attraverso una tecnica differente.

In molte delle immagini che presentano il problema descritto precedentemente, una piccola porzione dei pixel situati all'interno della sclera ha comunque una tonalità blu, ma purtroppo si tratta di un numero estremamente basso di pixel che non incide ovviamente sull'istogramma che calcoliamo.

Inizialmente si è pensato di creare una maschera con colori molto chiari, ma tendenti all'arancione o al giallo, utilizzando, come per la maschera dei punti scuri, la funzione *inRange* di OpenCV. Successivamente però si è deciso di escludere un range ristretto sul valore di *hue* in modo tale da rendere il risultato che si ottiene indipendente dal colore della luce con cui è stata catturata l'immagine.

Questa volta i valori di l e r sono:

- $l = \langle 0, 0, 90 \rangle$
- $r = \langle 179, 90, 255 \rangle$

Questo range di valori è stato ottenuto sperimentalmente dopo aver visionato diversi valori di *saturation* e *value* per i pixel all'interno della sclera su immagini differenti con luci differenti.

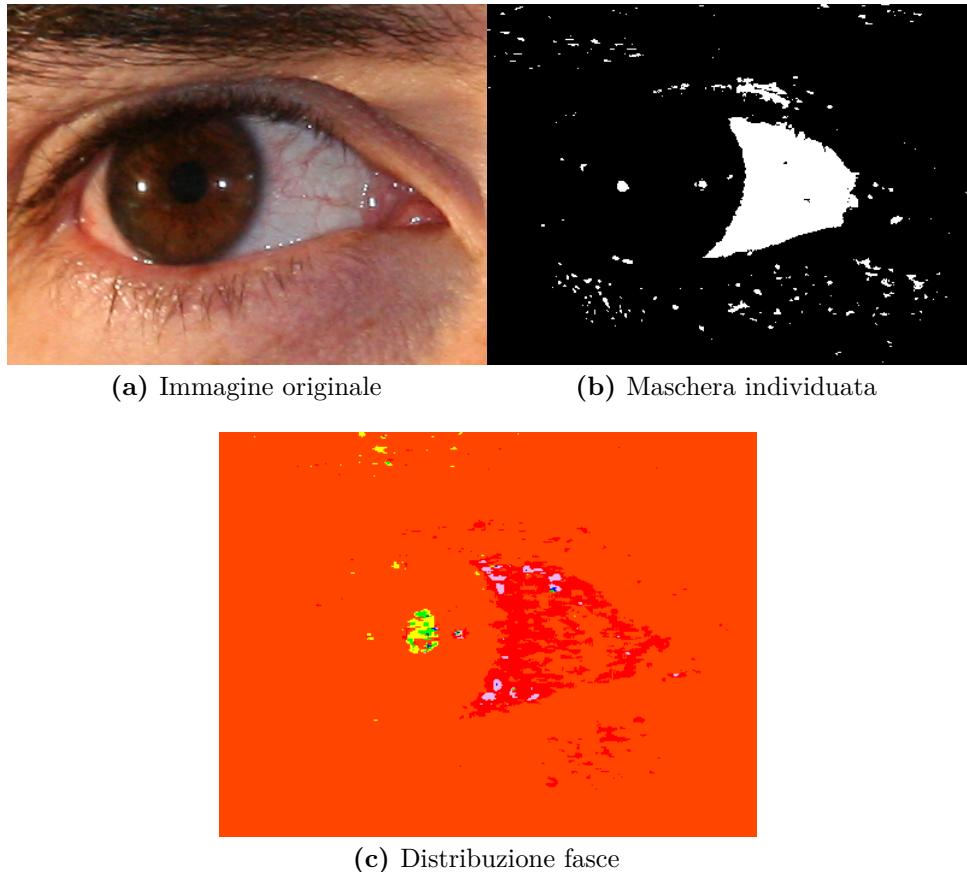


Figura 5.11

Andando a visionare le maschere ottenute su diverse immagini di test si può notare che:

1. La sclera viene individuata e alcune volte anche meglio rispetto all'utilizzo del blu (esempio in figura 5.11)
2. All'interno di queste maschere spesso vengono accesi anche dei pixel appartenenti alla pelle, non consentendoci di isolare la sclera dal resto (esempio in figura 5.12)

Per risolvere il problema appena descritto nel punto 2 sono state ideate due soluzioni che verranno descritte tra poco; tuttavia attualmente tra le due viene utilizzata solo la seconda proposta in quanto più efficiente ed efficace della prima; quest'ultima viene riportata comunque in quanto potrebbe essere migliorata e potrebbe risultare molto utile da utilizzare in combinazione con la seconda proposta. Entrambe si basano sulla creazione di nuove maschere che ci aiutano ad isolare al meglio i pixel appartenenti alla sclera.

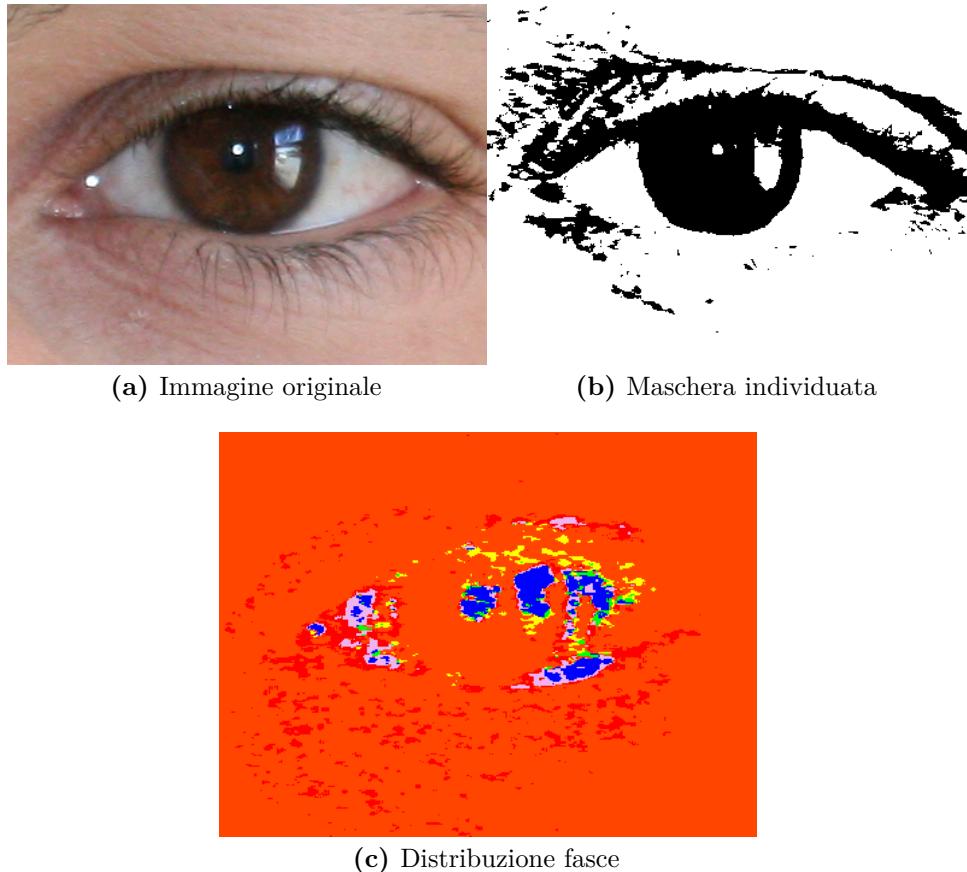


Figura 5.12

Prima soluzione

La prima soluzione è molto semplice perché consiste nel creare una nuova maschera *skinMask* la quale, come suggerisce anche il nome, evidenzi i pixel appartenenti alla pelle, così che mettendo la sua negazione in AND bitwise con la maschera della sclera individuata precedentemente otteniamo una nuova maschera in cui vengono evidenziati i soli pixel della sclera.

Per la creazione di *skinMask* è stata progettata una funzione *isSkinPixel* così definita:

$$isSkinPixel(rgb, hsv, ycrcb) = \begin{cases} true, & \text{se è un pixel della pelle} \\ false, & \text{altrimenti} \end{cases} \quad (5.5)$$

dove *rgb*, *hsv* e *ycrcb* sono le tre triple di valori per il pixel preso in esame sui tre spettri di colori.

La formula che viene utilizzata per dire se un pixel è appartenente alla pelle è presa da [10] ed è la seguente:

$$\begin{aligned} first = & (0 \leq h \leq 50) \wedge (0.23 \leq s \leq 0.68) \wedge \\ & (r > 95) \wedge (g > 40) \wedge \\ & (b > 20) \wedge (r > g) \wedge \\ & (r > b) \wedge (|r - g| > 15) \end{aligned}$$

$$\begin{aligned} second = & (r > 95) \wedge (g > 40) \wedge \\ & (b > 20) \wedge (r > g) \wedge \\ & (r > b) \wedge (|r - g| > 15) \wedge \\ & (cr > 135) \wedge (cb > 85) \wedge \\ & (y > 80) \wedge [cr \leq (1.5862 * cb) + 20] \wedge \\ & [cr \geq (0.3448 * cb) + 76.2069] \wedge \\ & [cr \geq (-4.5652 * cb) + 234.5652] \wedge \\ & [cr \leq (-1.15 * cb) + 301.75] \wedge \\ & [cr \leq (-2.2857 * cb) + 432.85] \end{aligned}$$

Con:

- $rgb = \langle r, g, b \rangle$, nella formula originale veniva incluso anche l'alpha channel, qui viene escluso in quanto è sempre pari a 1.
- $hsv = \langle h, s, v \rangle$, con $h \in [0, 360]$ ed $s, v \in [0, 1]$
- $ycrcb = \langle y, cr, cb \rangle$

ed infine:

$$isSkinPixel(rgb, hsv, ycrcb) = \begin{cases} true, & \text{se } first \vee second \\ false, & \text{altrimenti} \end{cases} \quad (5.6)$$

L'unica nota negativa di questa soluzione è che in alcuni casi tra i pixel rilevati ci sono quelli appartenenti alla sclera e quindi questo peggiora notevolmente i risultati.

Di seguito vengono riportati alcuni esempi di risultati:

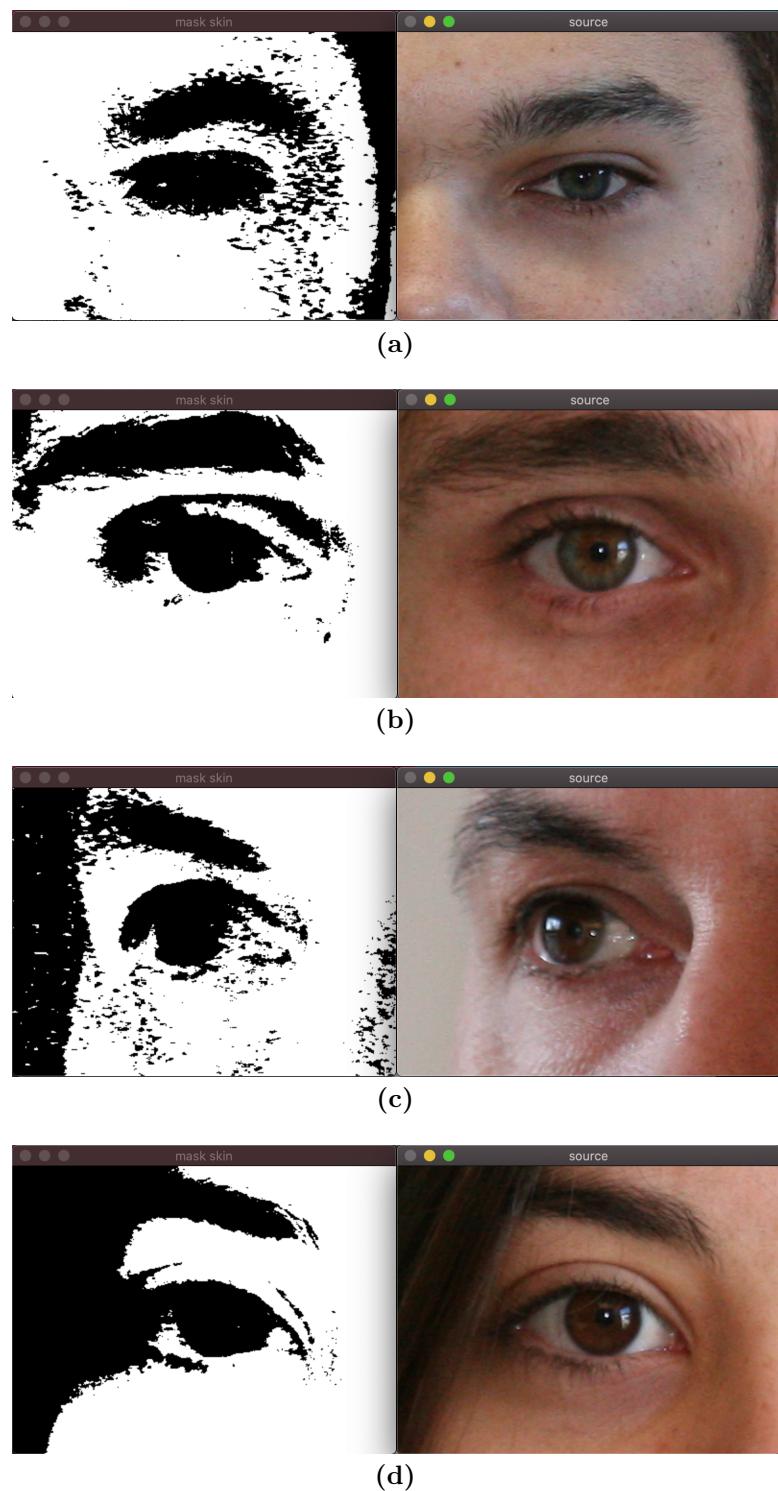


Figura 5.13

Seconda soluzione

La seconda soluzione che viene proposta e viene anche utilizzata attualmente all'interno della funzione *needClassifier* si basa sulla costruzione di una nuova maschera creata a partire da quella della sclera individuata precedentemente e dalla maschera B , ovvero quella che evidenzia i pixel aventi tonalità blu.

Sulla maschera della sclera vengono individuate diverse componenti connesse, le quali possono essere definite come porzioni di pixel accesi circondate da pixel spenti che le isolano dalle altre componenti (esempio in figura 5.14). Successivamente per ogni componente si controlla che ci sia al suo interno almeno un pixel che è acceso anche nella maschera B : in questo caso tutti i pixel di quella componente vengono accesi nella maschera restituita in output.

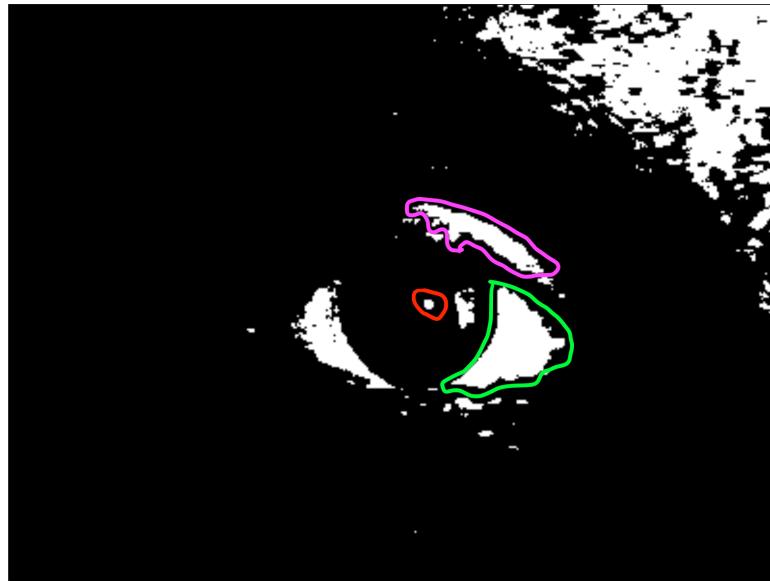


Figura 5.14. In rosso, viola e verde vengono evidenziate alcune componenti connesse

Un altro modo di vedere questo algoritmo consiste nel trasformare la maschera della sclera in un grafo G non diretto così definito:

- $G = (V, E)$
- $V =$ insieme dei pixel accesi nella maschera
- $E = \{(px_1, px_2) \mid \forall px_1, px_2 \in V \text{ t.c. } px_1 \text{ è adiacente a } px_2\}$

Successivamente viene applicato l'algoritmo di *Tarjan* [1] a G per il calcolo delle componenti connesse con l'unica differenza che ogni componente è individuata a partire dai nodi che corrispondono ad un pixel acceso nella maschera B , in modo tale da considerare soltanto le componenti che hanno all'interno un pixel che è acceso in B . Infine viene restituita in output una maschera che ha come pixel accesi soltanto quelli appartenenti alle componenti individuate.

La complessità di questo algoritmo viene approssimata a quella di *Tarjan* [1] che è lineare in base al numero di archi e nodi nel grafo, ossia $O(|V| + |E|)$.

Nell'implementazione che poi è stata effettuata, per ridurre il tempo di calcolo, non è stato effettuato alcun preprocessing sulla maschera della sclera per trasformarla in un grafo, ma si è proseguito con l'utilizzo della struttura a matrice a due dimensioni fornita da OpenCV, trattando i vari pixel accesi come nodi.

Di seguito in figura (5.15) possiamo vedere un esempio di applicazione della funzione:

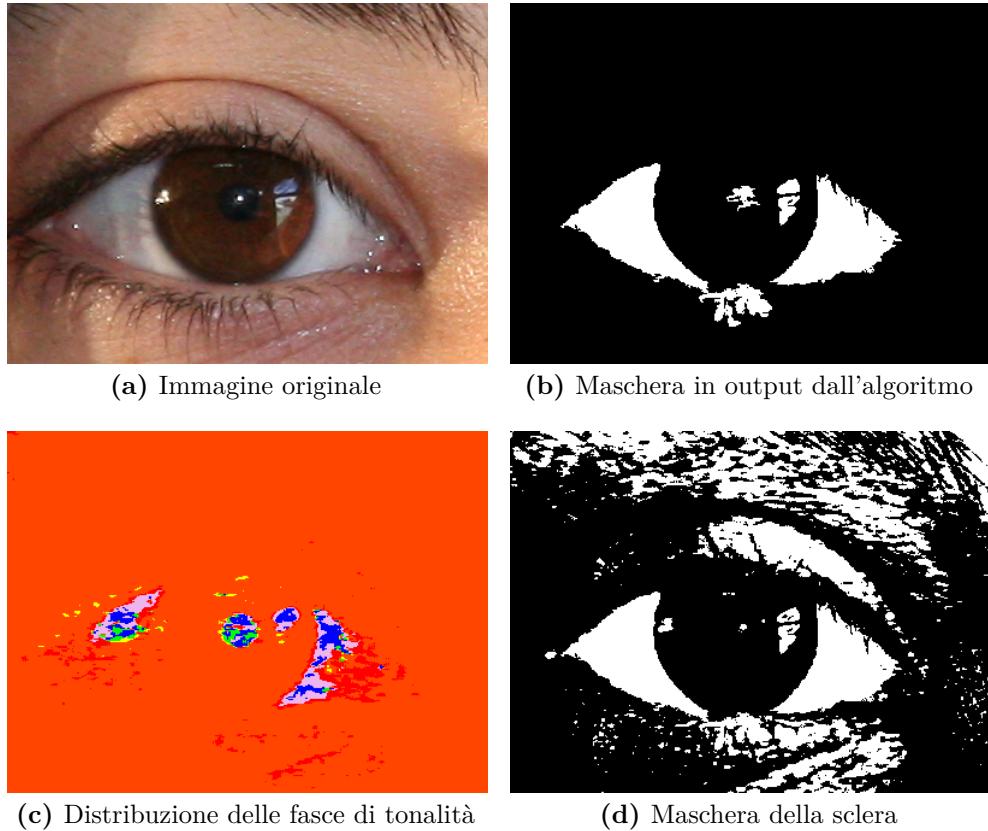


Figura 5.15

Come si può notare la sclera che nell'immagine (c) non viene quasi individuata, ora viene rilevata correttamente nell'immagine (b) utilizzando l'algoritmo proposto.

Definizione del controllo C_2

Definiamo il controllo C_2 come segue:

$$C_2 = freq_{Accesi} \geq t_2 \quad (5.7)$$

dove:

- $freq_{Accesi} = \frac{pixel_{Accesi}}{totPixel}$
- $pixel_{Accesi}$ è il numero di pixel che hanno valore pari a 255

- *totPixel* è il numero totale di pixel presenti nella maschera

Il valore di t_2 che è utilizzato attualmente è pari a 0.04 e rende $FAR(t) = FRR(t)$, ma nel caso in cui venissero implementati altri controlli questo potrebbe essere cambiato in modo tale da abbassare il FRR cosicché, nei casi più incerti, l'immagine presa in esame venga sottoposta ad ulteriori controlli.

5.7.4 Risultati ottenuti dalla funzione *needClassifier*

Una volta definiti i due controlli C_1 e C_2 con le rispettive soglie $t_1 = 0.03$ e $t_2 = 0.04$ si è proseguiti con l'analisi delle prestazioni di questa funzione.

Di seguito sono elencati i risultati ottenuti:

1. Per quanto riguarda I_{Near} :
 - (a) 3978 su 4460 (89%) non vengono passate al classificatore
 - (b) 482 su 4460 (11%) vengono passate al classificatore
2. Per quanto riguardo I_{Far} :
 - (a) 1737 su 6641 (26%) erroneamente non vengono passate al classificatore
 - (b) 4904 su 6641 (74%) vengono passate al classificatore

I casi in cui la funzione restituisce un output errato sono dati dai punti **1b** e **2a**, quindi si può affermare che su un totale di 11101 immagini la funzione restituisce un risultato corretto nell'80% dei casi.

Capitolo 6

Conclusioni

Nel presente lavoro di tirocinio sono stati descritti dettagliatamente i processi di segmentazione e riconoscimento dell'iride con le relative re-implementazioni migliorate degli algoritmi già presenti.

La prima implementazione, in Java, di *ISIS v.2* [12] presenta alcuni problemi per quanto riguarda la segmentazione in quanto questa, in alcuni casi non definiti, non avviene correttamente; sono state trovate delle soluzioni, come ad esempio l'utilizzo di un filtro di blur prima di posterizzare l'immagine, ma non sono stati trovati dei discriminanti che consentano l'automatizzazione di questo processo.

Nella vecchia implementazione, a differenza di quella corrente, gli algoritmi utilizzati non sono stati implementati efficientemente, come ad esempio l'operazione di posterizzazione la quale prevedeva per ogni pixel il calcolo di un nuovo istogramma sull'intera finestra utilizzata, generando così un costo computazionale alto; nella nuova implementazione invece è stata posta sin da subito una particolare attenzione ai costi computazionali cercando di ridurli al minimo possibile.

Un'altra problematica presente nell'implementazione precedente del metodo *ISIS v.2* riguarda la sua adattività, in quanto quando viene passata in input un'immagine che inquadra principalmente un occhio, questa viene comunque passata direttamente al classificatore il quale restituisce un output errato. In questo lavoro di tirocinio invece sono stati implementati diversi controlli a cascata che ci consentono di poter decidere quando un'immagine necessita del classificatore di occhi, con una percentuale di successo dell'80%.

6.1 Sviluppi futuri

Un possibile sviluppo futuro riguarda l'automatizzazione della scelta del metodo di thresholding da applicare per la rimozione dei riflessi e della scelta di utilizzo del filtro di blur, così da poter esonerare l'utente da queste due scelte.

Un secondo possibile sviluppo riguarda l'implementazione di ulteriori controlli da aggiungere alla funzione *needClassifier* così da ridurre ulteriormente l'errore del 20% ottenuto nella fase di testing. Un possibile controllo da aggiungere potrebbe prevedere l'utilizzo della maschera della pelle, la quale risulta molto utile per individuare sia i possibili pixel di sfondo di un soggetto, sia i pixel stessi della pelle, anch'essi da escludere.

Un altro possibile sviluppo riguarda l'utilizzo delle tonalità del verde e del giallo per poter individuare ed eventualmente scartare delle immagini con parecchio rumore, dato che, come è stato visto nella prima analisi massiva, in questo tipo di immagini queste tonalità sono molto presenti.

Bibliografia

- [1] Tarjan, R. E. (1972), *Depth-first search and linear graph algorithms*, SIAM Journal on Computing, 1 (2): 146–160, doi:10.1137/0201010
- [2] Canny, J. *A computational approach to edge detection*. IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-8 (1986), 679. doi: 10.1109/TPAMI.1986.4767851.
- [3] Daugman, J. G. *High confidence visual recognition of persons by a test of statistical independence*. IEEE Transactions on Pattern Analysis and Machine Intelligence, 15 (1993), 1148. doi:10.1109/34.244676.
- [4] Ojala, T., Pietikäinen, M., and Harwood, D. *A comparative study of texture measures with classification based on featured distributions*. Pattern Recognition, 29 (1996), 51 . Available from: <http://www.sciencedirect.com/science/article/pii/0031320395000674>, doi:[https://doi.org/10.1016/0031-3203\(95\)00067-4](https://doi.org/10.1016/0031-3203(95)00067-4).
- [5] Cole, S. *Suspect Identities, A History of Fingerprinting and Criminal Identification* (2001). ISBN 9780674010024.
- [6] Telea, A. *An image inpainting technique based on the fast marching method*. Journal of Graphics Tools, 9 (2004), 23. Available from: <https://doi.org/10.1080/10867651.2004.10487596>, arXiv:<https://doi.org/10.1080/10867651.2004.10487596>, doi:10.1080/10867651.2004.10487596.
- [7] Proenca, H., Filipe, S., Santos, R., Oliveira, J., and Alexandre, L. A. *The ubiris.v2: A database of visible wavelength iris images captured on-the-move and at-a-distance*. IEEE Transactions on Pattern Analysis and Machine Intelligence, 32 (2010), 1529. doi:10.1109/TPAMI.2009.66.
- [8] Haindl, M. and Krupička, M. *Accurate detection of non-iris occlusions*. Proceedings - 10th International Conference on Signal-Image Technology and Internet-Based Systems, SITIS 2014, (2015), 49. doi:10.1109/SITIS.2014.48.
- [9] Haindl, Michal e Mikuláš Krupicka (2015). *Unsupervised detection of non-iris occlusions*.
- [10] Kolkur, S. Kalbande, Dhananjay Shimpi, P. Bapat, C. Jatakia, Janvi. (2017). *Human Skin Detection Using RGB, HSV and YCbCr Color Models*. 10.2991/iccasp-16.2017.51. <https://doi.org/10.2991/iccasp-16.2017.51>
- [11] Riso, M. (2018). *Riconoscimento dell'iride tramite combinazione di operatori*.
- [12] Varone, V. (2019). *Miglioramento delle prestazioni dell'algoritmo di segmentazione dell'iride ISIS e confronto tramite IRISSEG*.

Ringraziamenti

Voglio dedicare questo spazio per ringraziare tutte le persone che mi sono state vicine con il loro supporto durante la realizzazione di questo elaborato.

Ringrazio la mia relatrice, la prof.ssa De Marsico per la sua grande disponibilità ad ogni mia richiesta e per i suoi preziosi consigli che mi hanno guidato nelle ricerche e nella stesura.

Ringrazio la mia famiglia la quale ha sempre creduto nelle mie capacità e durante questi tre anni mi è stata vicina dandomi sempre giusti consigli.

Ringrazio la mia fidanzata Martina per il tempo che mi ha dedicato e per tutto il sostegno che mi ha sempre dato; avere il suo supporto è stato indispensabile soprattutto in questo ultimo periodo per me molto difficile.

Ringrazio infine i miei colleghi, ma soprattutto amici, con cui ho condiviso questo splendido percorso.