ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

Scuola di Scienze
Dipartimento di Fisica e Astronomia
Corso di Laurea in Fisica

METODICHE DI STATISTICAL E MACHINE LEARNING PER ANALISI DI IMMAGINI MEDICHE

Relatore:
Prof. Gastone Castellani

Presentata da:
Laura Verzellesi

Anno Accademico 2019/2020

Sommario

Abstract

Questa tesi ha come obiettivo la creazione e l'implementazione di un codice Python per la classificazione dei tessuti all'interno di un'immagine medica in formato DCOM. A tale scopo, sono adottate le immagini mediche PET-CT fusion per il loro vantaggio nel fornire informazioni sia anatomiche che funzionali. Sono complessivamente importate e analizzate 6 immagini (set di slices): 5 di queste sono utilizzate per il labelling e la rimanente parte è usata per la predizione del modello. Dopo una breve descrizione delle funzioni matematiche alla base del calcolo ed estrazione delle features di Haralick (la matrice di co-occorrenza, misura posizionale dei livelli di grigio di un'immagine, e le 14 features di Haralick), viene presentata in dettaglio l'implementazione di tali funzioni nel codice e le modalità con cui sono utilizzate quest'ultime con la finalità di riconoscere i diversi tipi di tessuti. Sono illustrati gli step progettuali della creazione dei dataset delle label e delle features e della selezione e allenamento del miglior modello per la classificazione. L'approccio seguito permette di classificare diversi pixel di un'immagine DCOM nelle classi "bone", ossa, "organ", organi, e "background", sfondo. Possibili future implementazioni della metodologia adottata comprendono la creazione di nuove classi per affinare la classificazione dei diversi tessuti.

Introduzione

L’elaborato descrive la creazione e l’implementazione di un codice Python in grado di classificare diversi tessuti all’interno di un’immagine in formato DCOM. Nel primo capitolo è illustrato brevemente l’ambiente di lavoro, dunque, il linguaggio di programmazione scelto e il notebook utilizzato. Inoltre, ci si è concisamente soffermati sulla libreria SimpleITK utilizzata per gestire le immagini mediche in formato DCOM e sull’applicazione ITK-snap usata per visualizzare le immagini DCOM in 3D. Il secondo capitolo chiarifica i principali vantaggi delle immagini mediche PET-CT fusion nell’apporto di informazioni anatomiche e funzionali e illustra le modalità utilizzate per importare, leggere ed elaborare le immagini in formato DCOM con Python. Il terzo capitolo ha lo scopo di mostrare le funzioni matematiche alla base del calcolo ed estrazione delle features di Haralick per il perseguimento dell’obiettivo di classificazione. Sono definite la matrice di co-occorrenza e le 14 features di Haralick ed è presentata la modalità di implementazione di tali funzioni nel codice. Nel quarto capitolo sono riportati gli step progettuali dalla creazione dei dataset delle label e delle features alla selezione e allenamento del miglior modello per la classificazione dei diversi tessuti. In ultimo, nel quinto capitolo sono descritti le possibili migliorie apportabili al codice per affinare la classificazione.

Capitolo 1. Ambiente di lavoro

1.1 Python e Jupyter

Python è un linguaggio di programmazione ad alto livello: distaccandosi dal linguaggio macchina risulta facilmente interpretabile dall'utente. Progettato nel 1991 dal programmatore olandese Guido Van Rossum e sviluppato dalla Python Software Foundation, supporta diversi paradigmi di programmazione: Object-Oriented, che permette di definire degli oggetti software in grado di interagire e comunicare tra di loro; funzionale, in cui il flusso di esecuzione del programma assume la forma di una serie di valutazioni di funzioni matematiche; procedurale, basata sull'esecuzione di una serie di passaggi computazionali.

Offre un gran numero di librerie built-in, che insieme alla gestione degli errori molto chiara ed efficiente fa di Python uno dei linguaggi di programmazione più comodi e facili da usare.

Il codice risulta molto chiaro e facile da leggere, infatti, a differenza di molti altri linguaggi, non sono utilizzate parentesi per delimitare i blocchi logici (cicli for e while, costrutti condizionali...) ma si fa uso delle indentazioni, il punto e virgola al termine di un'istruzione è facoltativo, le variabili non sono tipizzate e sono presenti meno eccezioni sintattiche. Tutte le variabili sono in realtà puntatori a oggetto (reference): a una variabile a cui è stato assegnato un valore di tipo intero, può essere assegnata subito dopo una stringa o una lista.

Un'altra caratteristica importante è che è un linguaggio pseudocompilato, il che vuol dire che tra l'uomo e il linguaggio macchina si interpone un "traduttore", che analizza il file sorgente e, una volta assicuratosi che sia sintatticamente corretto, si occupa di eseguirlo. L'essere pseudointerpretato rende Python interfacciabile con un qualsiasi sistema operativo, infatti una volta scritto un sorgente esso può essere interpretato dalla maggior parte dei sistemi, quali Mac, Microsoft Windows e GNU/Linux.

Il notebook Jupyter è un'applicazione che consente la visualizzazione, la modifica e l'esecuzione di documenti notebook tramite un web browser. I documenti notebook sono documenti che contengono sia il codice di programmazione sia elementi come testi, equazioni figure, tabelle, links, etc. Per la combinazione di codice ed elementi di testo, questi documenti sono il luogo ideale per eseguire il codice creato e riportare la descrizione dell'analisi compiuta e i relativi risultati.

1.2 SimpleITK e ITK-snap

La piattaforma Insight Segmentation and Registration Toolkit (ITK), sviluppata con il finanziamento della United States National Library of Medicine, fornisce una collezione di algoritmi per la segmentazione (processo di identificazione e classificazione dei dati di un'immagine digitale) e la registrazione in più dimensioni (allineamento o sviluppo di corrispondenze tra i dati: ad esempio, in ambito medico, un'immagine CT può essere allineata e combinata con un'immagine MRI per ottenere maggiori informazioni). La libreria di analisi immagini SimpleITK consiste in un'interfaccia open source semplificata della piattaforma ITK.

La libreria è implementata in C++ ma grazie al processo di wrapping automatico genera interfacce tra C++ e altri linguaggi di programmazione come Python, Java, R. Ciò consente agli sviluppatori di creare software utilizzando una grande varietà di linguaggi. Essendo un progetto open source, sviluppatori da tutto il mondo possono utilizzare, gestire, estendere, correggere il software.

L'obiettivo primario della libreria è quello di rendere gli algoritmi accessibili alla più ampia gamma di scienziati possibile. L'obiettivo secondario è quello di promuovere la combinazione tra gli strumenti di analisi di immagini della libreria stessa e gli strumenti computazionali disponibili in linguaggi come Python o R. Questo, insieme con l'interfaccia user friendly, rende la libreria Simple ITK frequentemente utilizzata in molte applicazioni per l'analisi di immagini mediche.

Nel codice creato la libreria è stata importata per gestire l'importazione delle immagini DCOM.

ITK-snap è un'applicazione software interattiva utilizzata per visualizzare le immagini mediche tridimensionali, delineare manualmente le regioni anatomiche di interesse ed eseguire la segmentazione automatica delle immagini. È il prodotto della collaborazione tra Paul Yushkevich, dottorato di ricerca presso l'università della Pennsylvania, e Guido Gerig, dottorato di ricerca presso l'università dello Utah. ITK-snap è uno strumento gratuito, open source e multipiattaforma, che sfrutta la libreria ITK e presenta molteplici funzionalità come la segmentazione manuale contemporaneamente in tutti e tre i piani ortogonali e il supporto per diversi formati di immagini (DCOM, NIfTI, Mayo Analyze).

Capitolo 2. Immagini DCOM

2.1 Le immagini PET-CT fusion

La tomografia a emissione di positroni (PET) è una tecnica di imaging che permette una valutazione quantitativa non invasiva dei processi biochimici e funzionali del corpo umano. La tecnica si basa sulla determinazione della distribuzione di concentrazione di un particolare radionuclide contenuto in un radiofarmaco iniettato o fatto ingerire preventivamente al paziente. I radiofarmaci hanno come target i tessuti di cui si vuole evidenziare lo stato fisio-patologico e, una volta all'interno dell'organismo, determinano l'emissione di positroni β^+. Dopo un breve percorso il positrone si annichila con un elettrone del tessuto, generando una coppia di fotoni γ co-lineari che vengono rilevati da due schermi posti in posizioni opposte. Un tale evento permette di individuare la linea di risposta (LOR) lungo la quale si trova il nucleo emettitore del positrone, ovvero la linea lungo la quale si può retroproiettare l'evento per ottenere l'immagine voluta.
Il radiofarmaco più utilizzato è il fluorodeossiglucosio (FDG), composto dal radioisotopo ^{18}F legato chimicamente a una molecola metabolicamente attiva. L'FDG viene captato dalle cellule ad alto utilizzo di glucosio. Si accumula, quindi, fisiologicamente in molti organi sani tra cui il cervello, i muscoli, le ghiandole salivari, il miocardio, la ghiandola tiroidea, gli organi del tratto gastrointestinale e del tratto urinario. Ma si raccoglie in grande quantità anche nelle cellule tumorali. Come per il glucosio, all'ingresso nella cellula il FDG viene fosforilato in posizione 6, impedendone la fuoriuscita dalla cellula. A differenza del glucosio, tuttavia, l'FDG non può essere catabolizzato nella via glicolitica e rimane nella forma di FDG-6-fosfato fintantoché la molecola rimane radioattiva. Prima del decadimento, infatti, la molecola non può essere utilizzata a causa dell'ingombro sterico generato dal fluoro. La molecola prodotta dal decadimento, invece, risulta essere una vera e propria molecola di glucosio-6-fosfato, normalmente metabolizzabile dall'organismo [1].
In assenza di una correlazione anatomica che consenta di delineare le strutture interne del corpo umano, i siti patologici di accumulo del radiofarmaco FDG possono essere facilmente confusi con il normale assorbimento fisiologico, portando a risultati falsi positivi o falsi negativi. Ciò può costituire un importante limite nella determinazione del sito malato, in particolare per piccole lesioni o per lesioni situate vicino a siti di assorbimento fisiologico. Per questo, l'interpretazione di immagini tomografiche a emissione di positroni in assenza delle correlate informazioni anatomiche può essere impegnativa.

La tomografia computerizzata (CT) è una tecnica di imaging che utilizza un fascio collimato di raggi X per effettuare la scansione di un oggetto. Conoscendo il coefficiente di attenuazione e l'intensità del fascio prima e dopo l'attraversamento dell'oggetto in esame, è possibile calcolare il profilo di attenuazione che il fascio subisce nel passaggio. I dati relativi all'attenuazione del fascio rappresentano la proiezione dell'oggetto interposto tra la fonte e il rilevatore di raggi X. Per ricostruire l'immagine si utilizza il metodo di retroproiezione: ogni punto acquisito viene retroproiettato attraverso algoritmi matematici che rientrano nella categoria delle tecniche di Fourier. Si ottengono così immagini ad alta risoluzione estremamente utili per una conoscenza atomica approfondita.

La PET-CT fusion è una tecnologia che permette la correlazione dei risultati di due modalità di imaging simultanee in un esame completo: la CT mostra dettagli atomici ad alta risoluzione ma non fornisce informazioni funzionali, la PET rivela gli aspetti dei processi funzionali e permette misurazioni metaboliche ma non prevede punti di riferimento anatomici per un preciso orientamento morfologico. Di conseguenza, la PET-CT fornisce una definizione anatomica più precisa sia per l'assorbimento fisiologico che patologico del radiofarmaco FDG. Per la maggiore sensibilità e specificità delle informazioni fornite rispetto alle due tecniche separate, la combinazione PET-CT è particolarmente utilizzata nella diagnosi e nella cura dei tumori, ma è usata anche per molte altre patologie.

Nel periodo post terapia, ad esempio, attività metaboliche quasi impercettibili rilevate con PET che normalmente sarebbero trascurate, possono essere identificate come un residuo della malattia una volta correlata l'informazione ottenuta con i dati acquisiti per mezzo della CT. Allo stesso tempo, risultati equivoci della CT possono essere meglio valutati con l'aiuto delle ulteriori informazioni funzionali fornite dalla PET.

Per sopperire alle difficoltà tecniche e logistiche causate dallo spostamento del paziente per eseguire i due esami e alle imprecisioni conseguenti, si utilizzano i tomografi PET-TC, nei quali sono assemblati insieme il sistema di rilevazione PET e un tomografo TC di ultima generazione. L'introduzione del tomografo PET-TC ha consentito un grande miglioramento dell'accuratezza e dell'interpretabilità delle immagini ed una notevole riduzione dei tempi di esame.

La Figura 1, presa da un articolo della rivista RadioGraphics [2], mostra un confronto tra un'immagine CT, un'immagine PET e un'immagine data dalla combinazione delle due.

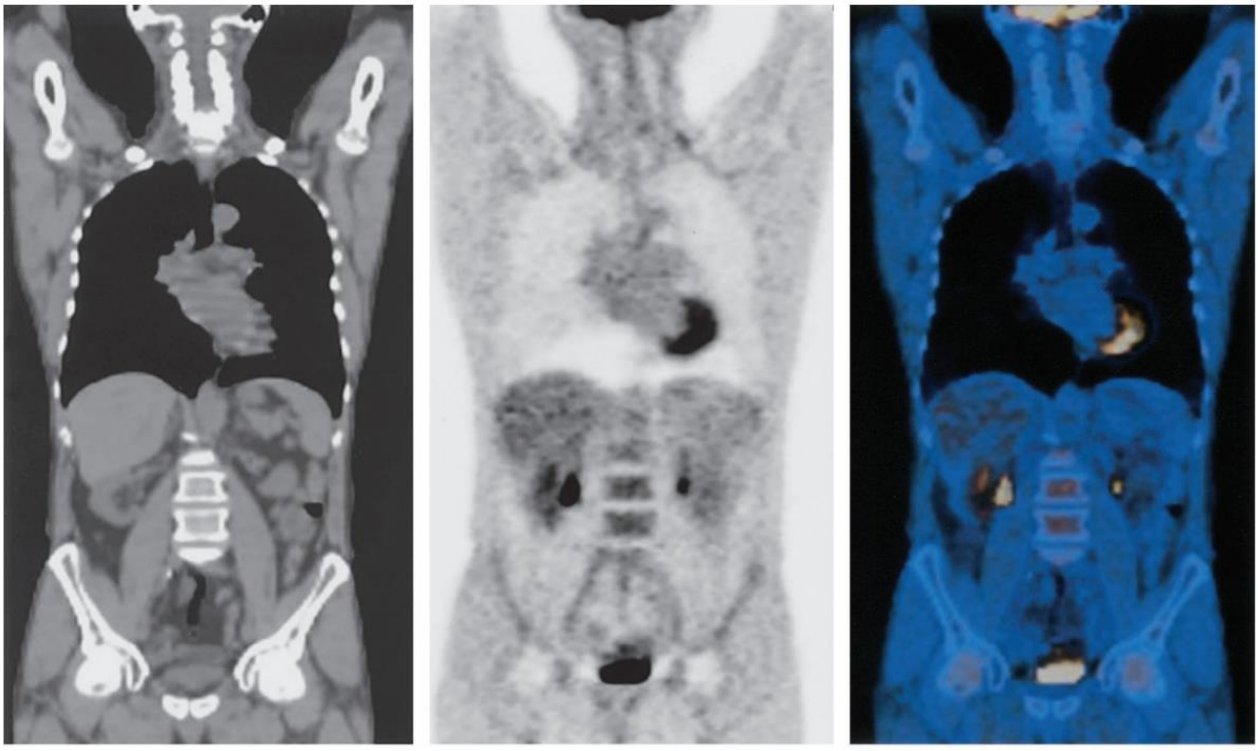

Figura 1. Confronto tra immagini ottenute per mezzo di diverse tecniche. L'immagine a sinistra è una CT, l'immagine al centro è una PET e l'immagine a destra una PET-CT fusion. La prima immagine mostra con accurata risoluzione l'anatomia interna del corpo umano. Dalla PET si nota il normale assorbimento del radiofarmaco FDG nella corteccia cerebrale-cerebellare alla base del cranio, nel miocardio, nel fegato, nei reni, nella vescica.

2.2 Lettura immagini DCOM con Python

Le immagini analizzate sono state scaricate dal database TCIA (The Cancer Imaging Archive). Sono state scelte immagini PET-CT fusion di due pazienti, ottenute da esami in date diverse. Complessivamente sono state analizzate 6 immagini (set di slices): 5 di queste sono state utilizzate per il labelling e la rimanente è stata usata per la predizione del modello.

Le immagini DCOM scaricate sono state importate sotto forma di array (*images_array*) attraverso la riga di codice seguente:

images_array = np.asarray(read_multiple_dcm(dir_path)),

dove *np.asarray()* è la funzione che permette di trasformare in array il suo argomento; *dir_path* è il percorso della directory locale in cui sono state salvate le immagini e *read_multiple_dmc(),* riportata di seguito, è la funzione utilizzata per leggere un set di immagini DCOM.

def read_multiple_dcm(dir_path):

```
reader = sitk.ImageSeriesReader()
dicom_names = reader.GetGDCMSeriesFileNames(dir_path)
reader.SetFileNames(dicom_names)
image = reader.Execute()
image_size = image.GetSize()
image_number = image_size[2]
images_array = []
for image_id in range(image_number):
    images_array.append(sitk.GetArrayFromImage(image[:,:,image_id]).squeeze())
return images_array
```

L'array *images_array* risulta avere dimensioni [275, 512, 512, 3]: consiste in un insieme di 275 immagini RGB (3 canali di colore: rosso, verde e blu), ciascuna di dimensioni 512x512 pixels.

2.3 Elaborazione delle immagini DCOM: tecniche di preprocessing

Prima di procedere con l'analisi delle immagini importate, sono state utilizzata due tecniche di preprocessing dell'immagine: la trasformazione dalla scala di colori RGB alla scala di grigi e la normalizzazione dell'immagine.

Per trasformare ciascuna immagine importata dalla sua rappresentazione per mezzo dei tre canali RGB alla rappresentazione in scala di grigi, è stata utilizzata la funzione *color.rgb2gray()*. Prima di poter applicare tale funzione, è stato necessario eseguire il reshape dell'array *images_array* con la seguente riga di codice:

images_array_reshaped = np.transpose(images_array, (2, 1, 0, 3)),

che scambia la prima dimensione (in posizione 0) di *images_array* con la terza (in posizione 2). In questo modo l'array *images_array_reshaped* ha dimensioni [512, 512, 275, 3]. È ora possibile applicare a *images_array_reshaped* la funzione *color.rgb2gray()*:

image_rgb = color.rgb2gray(images_array_reshaped)

per ottenere un array *image_rgb* di dimensioni [512, 512, 275].

La tecnica scelta per la normalizzazione dell'immagine si basa sul riscalare il valore di ciascun pixel dell'immagine nell'intervallo di valori impostato tramite la variabile *levels*. La funzione utilizzata è la seguente:

```
def normalize(img, levels = 16):
    img = (img-img.min())/(img.max()-img.min())
    img = (img*(levels-1)).astype(int)
    return img.
```

La funzione *normalize()* riceve come argomenti l'immagine da normalizzare e il numero di livelli scelti per la normalizzazione. Ad ogni pixel dell'immagine viene sottratto il minimo dell'immagine stessa (*img.min()*) e il risultato è successivamente diviso per la differenza tra il massimo e il minimo dell'immagine (*img.max()-img.min()*). Il valore appena ottenuto viene quindi moltiplicato per *levels*-1 = 15, avendo scelto 16 livelli ed essendo essi numerati a partire da 0 (da 0 a 15).

In Figura 2 è riportata la slice 107 dell'immagine dopo la fase di trasformazione in scala di grigi e dopo la fase di normalizzazione.

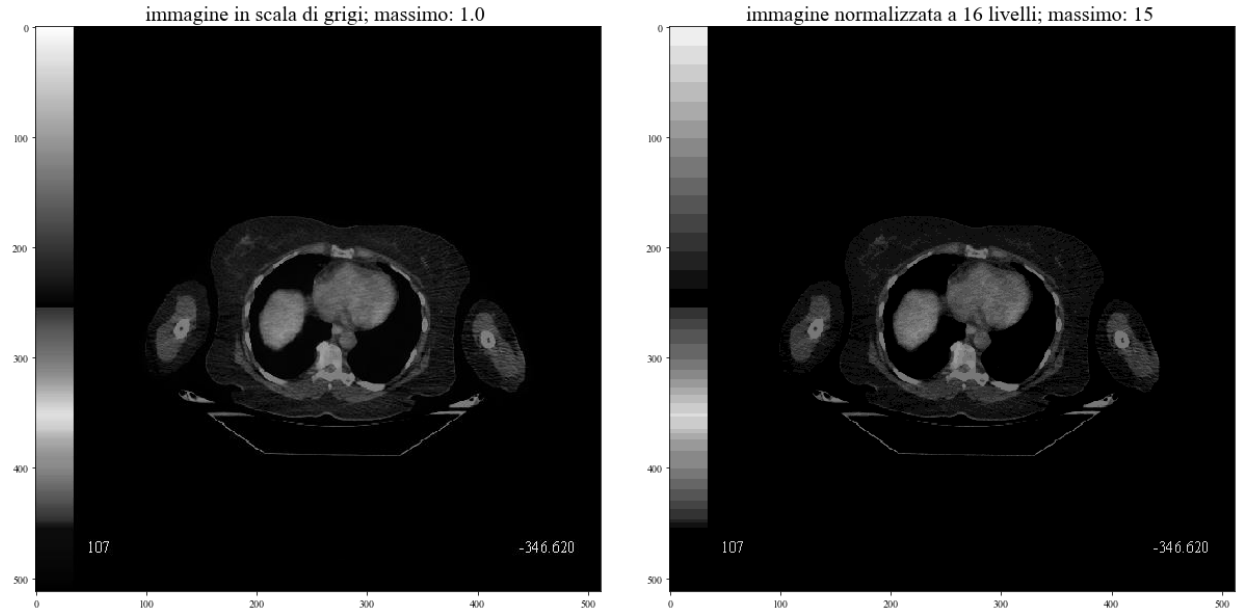

Figura 2. A sinistra è riportata la slice 107 l'immagine trasformata in scala di grigi, a destra è riportata la stessa slice dopo aver eseguito l'operazione di normalizzazione. Come si nota dal titolo, nel primo caso il pixel massimo ha valore 1, nel secondo ha valore 15.

L'immagine finale in scala di grigi normalizzata su 16 livelli viene assegnata alla variabile *image* ed è ora pronta per essere analizzata.

Capitolo 3. Calcolo GLCM ed estrazione features di Haralick

3.1 Matrice di co-occorrenza

La matrice di co-occorrenza GLCM è calcolata a partire da un'immagine I di dimensioni $n \times m$ determinando quante volte un pixel con intensità i compare in una determinata relazione spaziale con un pixel di valore j. Indicando con (Δx, Δy) la relazione spaziale tra il pixel di interesse e il suo intorno, la matrice di co-occorrenza è definita come:

$$GLCM_{\Delta x,\Delta y}(i,j) = \sum_{x=1}^{n}\sum_{y=1}^{m} \begin{cases} 1, & se\ I(x,y) = i\ \wedge I(x+\Delta x, y+\Delta y) = j \\ 0, & altrimenti \end{cases}$$

L'offset (Δx, Δy) risulta dalla scelta di un set di distanze (la tipica distanza è 1, il pixel adiacente) e di direzioni (0°, 45°, 90°, 135°). Ciascun elemento (i, j) nella matrice risultate è pari alla somma delle occorrenze nelle quali il pixel i è in relazione col pixel j nell'immagine analizzata. La mostra in dettaglio il processo di generazione di quattro matrici di co-occorrenza a partire da un'immagine di input. Sono utilizzati N=5 livelli di grigio e quattro differenti offset, definiti a partire dalle quattro possibili direzioni attorno al pixel selezionato (0°, 45°, 90°, 135°). Ad esempio, il pixel di valore 0 risulta per tre volte adiacente al pixel di valore 3, relazione spaziale definita dalla distanza 1 e dall'angolo 0° ed evidenziata dal riquadro di colore azzurro. L'occorrenza di tale fenomeno è dunque 3, come riportato nel punto di coordinate (0, 3) della prima matrice di co-occorrenza. Il pixel di valore 4 presenta solo una volta un pixel di valore 2 a distanza 1 e direzione 90°, relazione spaziale mostrata dal riquadro rosso. Il valore (4, 2) della corrispondente matrice di co-occorrenza risulterà, dunque, essere 1. Allo stesso modo sono calcolate le altre matrici rappresentate in Figura 3.

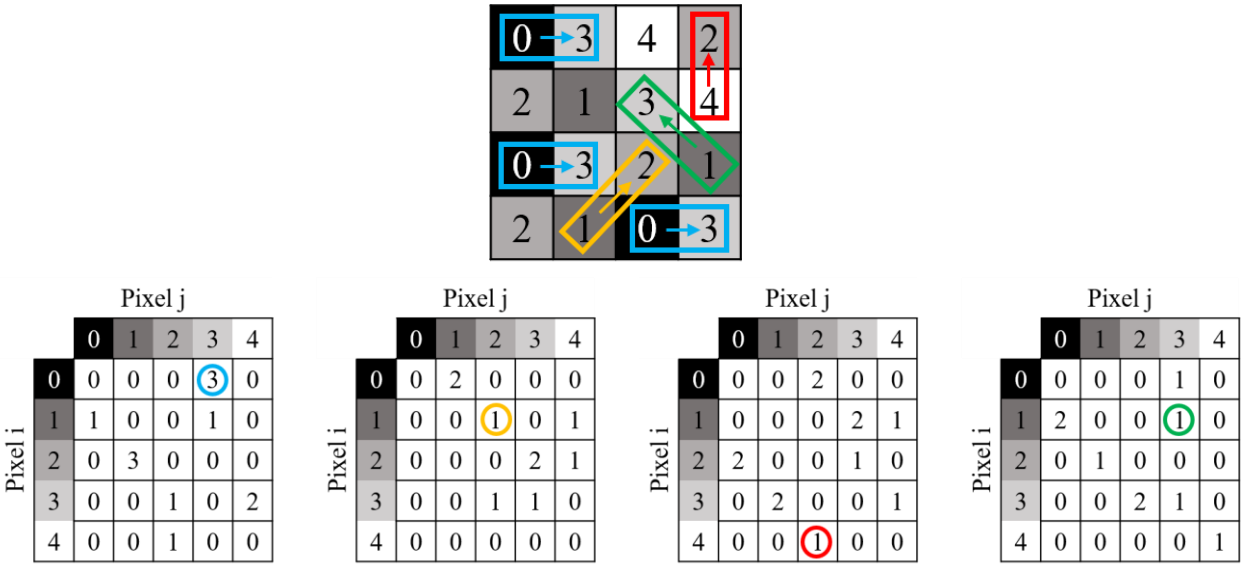

Figura 3. Rappresentazione del calcolo della matrice di co-occorrenza. L'immagine in alto rappresenta l'immagine di input, in cui sono specificati i valori del livello di grigio di ogni pixel. Le quattro immagini sottostanti corrispondono alle quattro matrici di co-occorrenza per N=5 livelli di grigio e 4 offset differenti: distanza 1 e angoli, rispettivamente, di 0°, 45°, 90° e 135°. In azzurro è evidenziata la relazione tra il pixel di valore 0 e il pixel di valore 3 a distanza 1 e angolo 0°; in giallo è evidenziata la relazione tra il pixel di valore 1 e il pixel di valore 2 a distanza 1 e angolo 45°; in rosso è evidenziata la relazione tra il pixel di valore 4 e il pixel di valore 2 a distanza 1 e angolo 90°; in verde è evidenziata la relazione tra il pixel di valore 1 e il pixel di valore 3 a distanza 1 e angolo 135°.

Il calcolo delle texture di un'immagine è più performante se applicato a matrici simmetriche: matrici i cui elementi sono simmetrici rispetto alla diagonale principale. Per ottenere matrici simmetriche è necessario contare ciascuna occorrenza tra una coppia di pixel due volte: una volta "in direzione" e una volta "nella direzione opposta". Ciò corrisponde a considerare un ulteriore set di direzioni (180°, 225°, 270°, 315°). Così

facendo, ad ogni iterazione, verrà contato quante volte ciascuna coppia di pixel si trova in una determinata relazione spaziale definita da una distanza 1 e da un set di angoli dato da: (0°-180°, 45°-225°, 90°-270°, 135°-315°). Dunque, il pixel di valore 0 risulta per tre volte adiacente al pixel di valore 3, relazione spaziale definita dalla distanza 1 e dall'angolo 0° e il pixel di valore 3 presenta per tre volte un pixel di valore 0 a distanza 1 e angolo 180°. Entrambi gli elementi (0, 3) e (3, 0) della corrispondente matrice di co-occorrenza avranno valore 3. Il procedimento è illustrato in Figura 4. In questo modo le matrici di co-occorrenza risultano simmetriche.

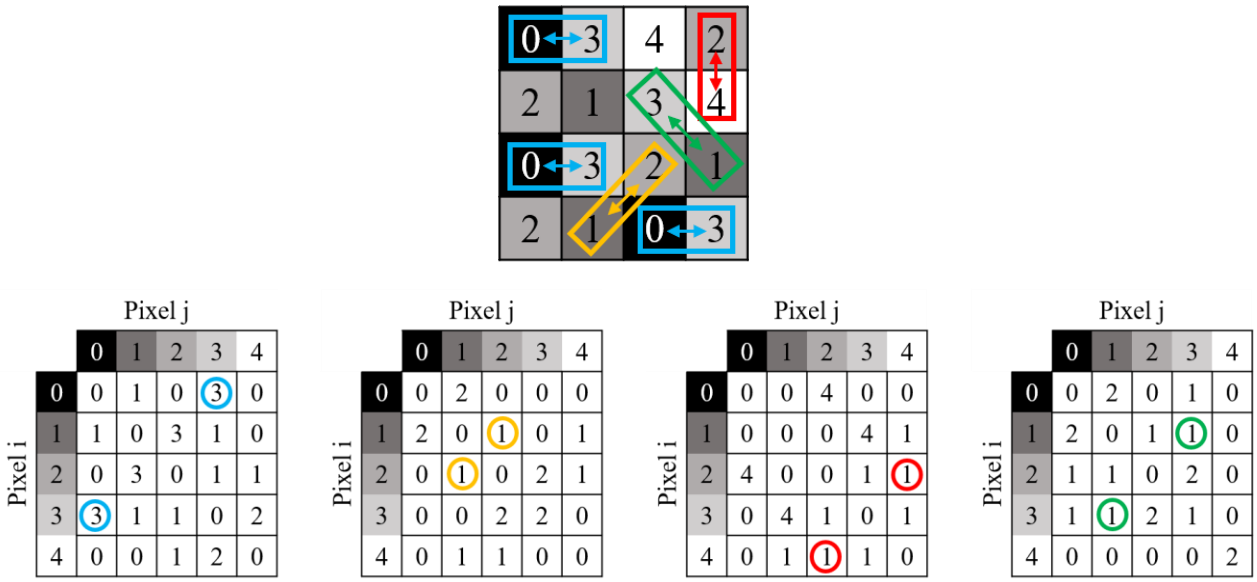

Figura 4. Rappresentazione del calcolo della matrice di co-occorrenza simmetrica. L'immagine in alto rappresenta l'immagine di input, in cui sono specificati i valori del livello di grigio di ogni pixel. Le quattro immagini sottostanti corrispondono alle quattro matrici di co-occorrenza per N=5 livelli di grigio e 4 offset differenti: distanza 1 e angoli, rispettivamente, di 0°-180°, 45°-225°, 90°-270° e 135°-315°. In azzurro è evidenziata la relazione tra il pixel di valore 0 e il pixel di valore 3 a distanza 1 e angolo 0°-180°; in giallo è evidenziata la relazione tra il pixel di valore 1 e il pixel di valore 2 a distanza 1 e angolo 45°-225°; in rosso è evidenziata la relazione tra il pixel di valore 4 e il pixel di valore 2 a distanza 1 e angolo 90°-270°; in verde è evidenziata la relazione tra il pixel di valore 1 e il pixel di valore 3 a distanza 1 e angolo 135°-315°.

Si può facilmente osservare che le matrici calcolate a partire dalle relazioni spaziali ottenute dai soli angoli (180°, 225°, 270°, 315°) risultano le trasposte delle matrici, riportate in Figura 3, calcolate a partire dalle relazioni spaziali ottenute dai soli angoli (0°, 45°, 90°, 135°). Dunque, per ottenere le matrici simmetriche in Figura 4 è sufficiente sommare alle matrici in Figura 3 le rispettive matrici trasposte, mantenendo così un solo set di angoli (0°, 45°, 90°, 135°).

Nel caso in cui l'immagine di input sia tridimensionale è sufficiente dichiarare un ulteriore set di angoli (0°, 45°, 90°, 135°) che corrisponderanno agli angoli nello spazio. Chiamando *angle_azim* il set di angoli nel piano, *angle_pol* il set di angoli nello spazio e *distance* la distanza preventivamente scelta (1 in questo caso), ed essendo *np.sin()* e *np.cos()* le funzioni matematiche per il calcolo di seno e coseno, è possibile ottenere la posizione relativa del pixel *j* rispetto al pixel *i* attraverso le seguenti relazioni:

*offset_row = int(distance * np.sin(angle_azim) * np.sin(angle_pol))*

*offset_col = int(distance * np.sin(angle_azim) * np.cos(angle_pol))*

*offset_dep = int(distance * np.cos(angle_azim))*

A questo punto, per ciascuna coppia possibile di angoli nel piano e nello spazio si calcolano le occorrenze dei valori di grigi tra ciascuna coppia di pixel, come precedentemente illustrato.

3.2 Calcolo della matrice di co-occorrenza per un'immagine DCOM 2D o 3D e visualizzazione heatmap

Dopo aver importato un'immagine DCOM come array (*images_array*) di dimensioni [275, 512, 512, 3] ed aver applicato la funzione *np.transpose(images_array, (1, 2, 0, 3))* per poter effettuare la trasformazione dai tre canali RGB a scala di grigi per mezzo della funzione *color.rgb2gray()*, si è ottenuto un array [512, 512, 275] su cui è poi stata operata la funzione di normalizzazione *normalize()*. Per poter calcolare le matrici di co-occorrenza dell'immagine in input si è scelto di selezionale un range di immagini dalle 275 importate e di ritagliarne solo una sezione, in quanto la computazione sull'intero set di slices sarebbe risultata estremamente lunga. Sono state selezionate sei slices 100x100 e pertanto l'array risulta di dimensioni [100, 100, 6]. Una slice di esempio è riportata in Figura 5.

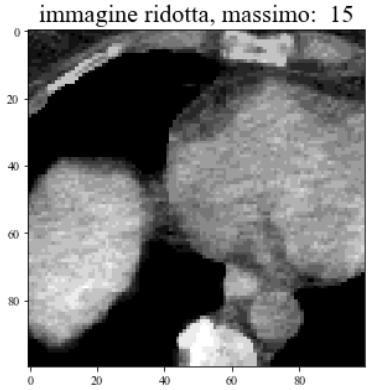

Figura 5. Esempio di una slice utilizzata per il calcolo della glcm. Per limitare i tempi computazionali è stata ritagliata una sezione 100x100 dall'immagine iniziale.

Per ogni angolo *angle_azim* e ogni angolo *angle_pol* viene chiamata la funzione *compute_glcm()* che riceve come argomenti l'immagine su cui calcolare le matrici di co-occorrenza, il numero di livelli di grigi dell'immagine, la distanza impostata a 1, un angolo dell'array *angle_azim* e un angolo dell'array *angle_pol*. La funzione *compute_glcm()*, sotto riportata, calcola le occorrenze di ciascuna coppia di pixel per ognuna delle 512 righe, ognuna delle 512 colonne e ognuna delle 6 slices dell'immagine importata. Avendo quattro angoli per ciascun array, si ottengono in questo modo 16 matrici di co-occorrenza. Ciascuna matrice così ottenuta è sommata alla sua trasposta attraverso la funzione *transpose()*, di seguito riportata, per trasformarla nella corrispondente matrice simmetrica.

```
def compute_glcm(img, levels, distance, angle_pol, angle_azim):
    rows = img.shape[0]
    cols = img.shape[1]
    dep = img.shape[2] #numero immagini
    out = np.zeros((levels, levels), dtype = np.uint32, order = 'C')
    offset_row = int(distance * np.sin(angle_azim) * np.sin(angle_pol))
    offset_col = int(distance * np.sin(angle_azim) * np.cos(angle_pol))
    offset_dep = int(distance * np.cos(angle_azim))
    start_row = max(0, -offset_row)
    end_row = min(rows, rows - offset_row)
```

```
    start_col = max(0, -offset_col)
    end_col = min(cols, cols - offset_col)
    start_dep = max(0, -offset_dep)
    end_dep = min(dep, dep - offset_dep)
    for r in range(start_row, end_row):
        for c in range(start_col, end_col):
            for d in range (start_dep, end_dep):
                i = img[r, c, d]
                row = r + offset_row
                col = c + offset_col
                dep = d + offset_dep
                j = img[row, col, dep]
                if 0 <= i < levels and 0 <= j < levels:
                    out[i, j] += 1
    return out.


def transpose_glcm(glcm):
    glcm_transp = glcm.transpose()
    glcm_def = glcm + glcm_transp
    return glcm_def
```

Le matrici ricavate vengono sommate per ottenere la matrice di co-occorrenza totale. Quest'ultima viene divisa per il numero totale di matrici calcolate e successivamente normalizzata per ridistribuire i valori dei pixel in un range tra 0 e 1. Per ottenere questo risultato, è sufficiente dividere la matrice totale ottenuta per la somma di tutti i valori della matrice totale.

```
glcm_tot = glcm_tot / (len(angles_pol)*len(angles_azim))
T = glcm_tot.sum()
glcm_tot_norm = glcm_tot / float(T)
```

La matrice GLCM normalizzata può essere interpretata come una funzione di probabilità delle coppie di livelli di grigi dell'immagine. È stato a questo punto possibile visualizzare la matrice di co-occorrenza totale e la corrispondente matrice di co-occorrenza normalizzata come heatmaps attraverso il codice seguente:

```
fig = plt.figure(figsize = (10,10))
plt.title('GLCM totale')
ax = sns.heatmap(glcm_tot, annot = True)
```

fig = plt.figure(figsize = (10,10))

plt.title('GLCM totale normalizzata')

ax = sns.heatmap(glcm_tot_norm, annot = True)

Le heatmaps ottenute sono mostrate in Figura 6 e Figura 7.

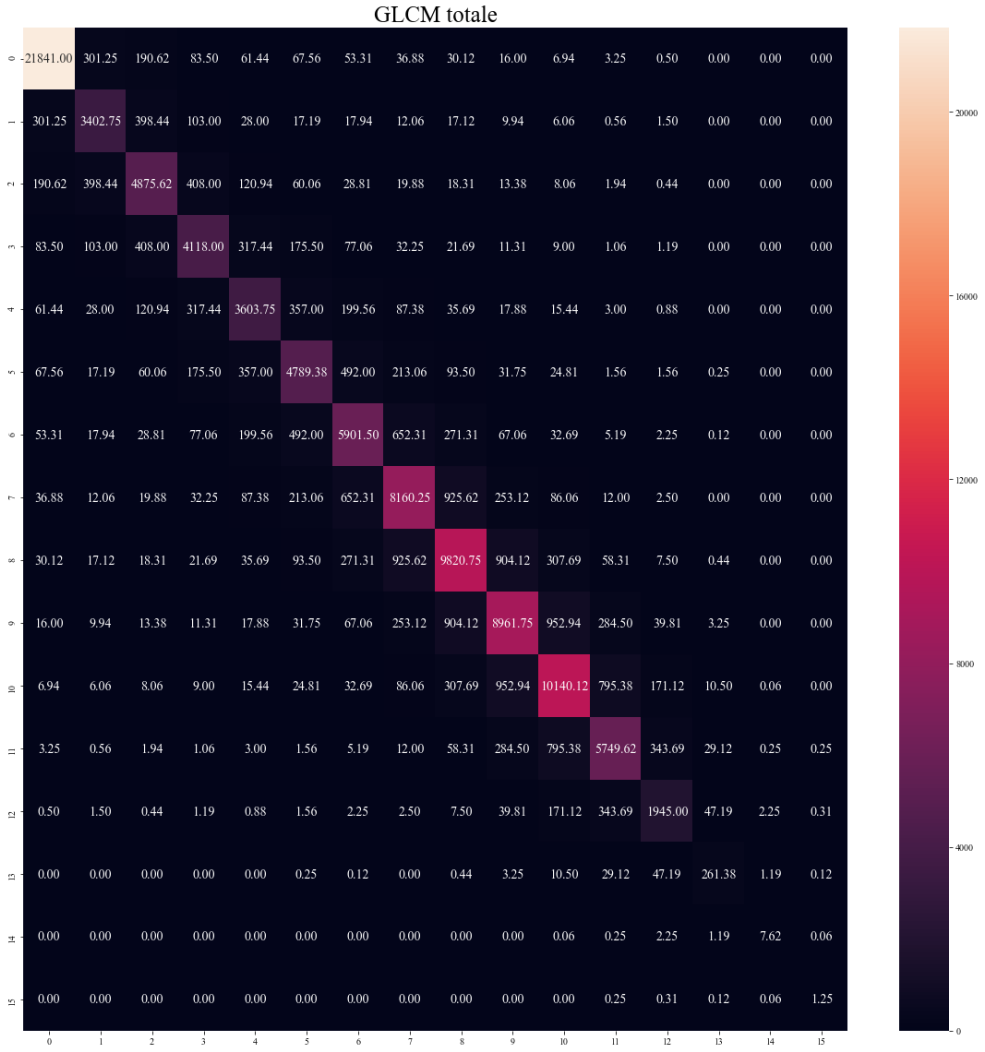

Figura 6. Matrice di co-occorrenza totale rappresentata come heatmap.

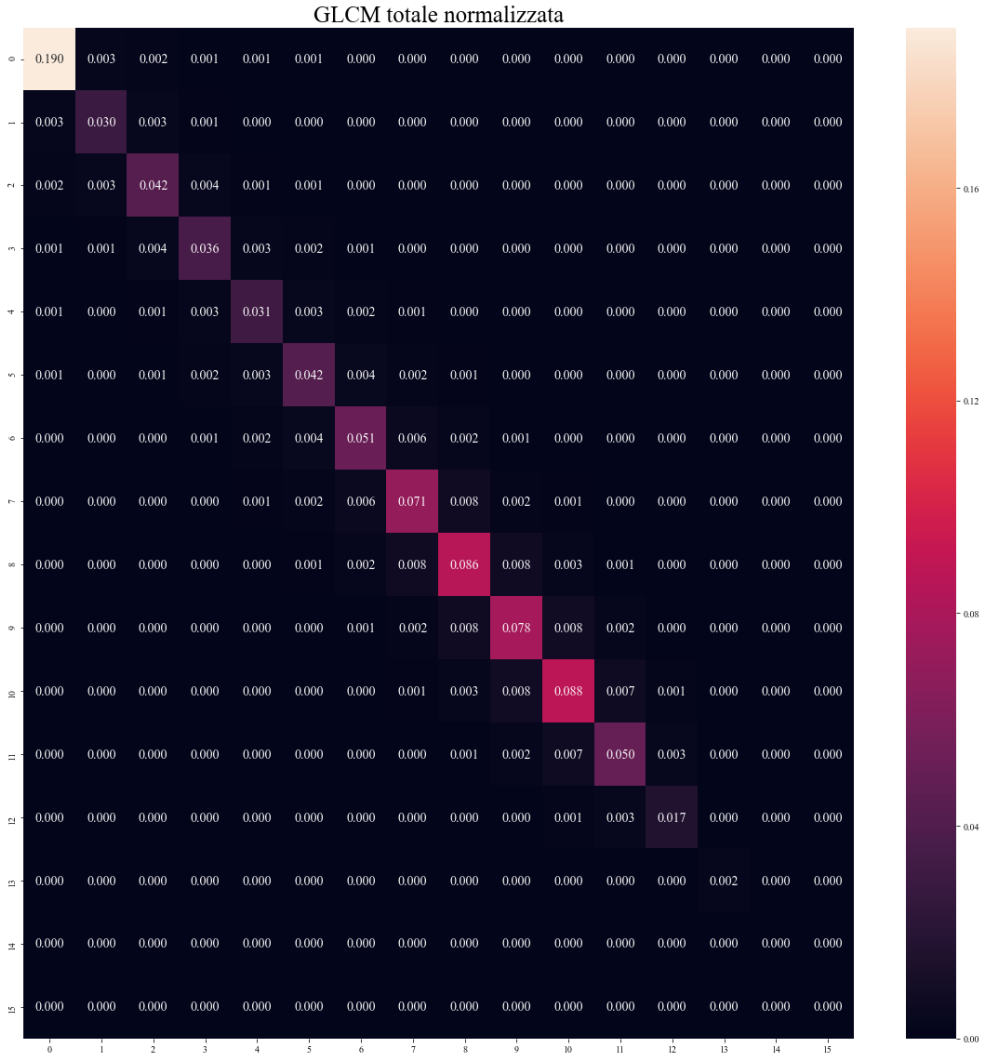

Figura 7. Matrice di co-occorrenza totale normalizzata rappresentata come heatmap.

3.3 Le 14 features di Haralick

Robert M. Haralick, nato nel 1943, è un famoso professore di Computer Science al Graduate Center of the City University of New York (CUNY). Haralick è una figura di rilievo nei campi del computer vision, pattern recognition, and image analysis. Nel 1973, Haralick ha introdotto 14 features statistiche comunemente utilizzate come descrittori delle texture di un'immagine. Nel suo articolo [3] Haralick afferma che le texture sono una proprietà intrinseca di qualsiasi superficie: il chicco di grano, la trama di una stoffa, il pattern delle colture nei campi, etc. Come tale, contengono importanti informazioni riguardo l'arrangiamento strutturale delle superfici e le loro relazioni con l'ambiente circostante. Nonostante per l'occhio umano sia facile riconoscere e tradurre in termini empirici le tipologie di texture, risulta estremamente più complicato far compiere la stessa azione a un digital computer. Siccome però, le texture di un'immagine procurano utili e considerevoli informazioni per scopi di classificazione e riconoscimento, è importante sviluppare delle funzioni per poter descrivere matematicamente le texture, rendendole così interpretabili da un digital computer.

La procedura per estrarre la texture da un'immagine si basa sull'assunzione che le informazioni sulla texture dell'immagine siano contenute nella relazione spaziale complessiva o media tra i toni di grigio nell'immagine stessa. Tale relazione spaziale è descritta dalla matrice di co-occorrenza precedentemente computata. Da quest'ultima è possibile estrarre le 14 textural features che descrivono informazioni come omogeneità, contrasto, entropia, varianza.

È innanzitutto fondamentale calcolare le variabili e le funzioni necessarie per ottenere le features di Haralick. Chiamando *p(i, j)* l'elemento *(i, j)* della matrice di co-occorrenza totale e normalizzata, e indicando con N il numero di livelli di grigio, si ottiene:

Marginal probabilities

$$p_x(i) = \sum_{j=1}^{N} p(i,j)$$

$$p_y(j) = \sum_{i=1}^{N} p(i,j)$$

Media

$$\mu_x = \sum_{i=1}^{N} i \cdot p_x(i)$$

$$\mu_y = \sum_{j=1}^{N} j \cdot p_y(j)$$

$$\mu = \frac{\mu_x + \mu_y}{2}$$

$$\mu_{x-y} = \sum_{k=0}^{N-1} k \cdot p_{x-y}(k)$$

Deviazione standard

$$\sigma_x = \sqrt{\sum_{i=1}^{N} p_x(i)(i - \mu_x)^2}$$

$$\sigma_y = \sqrt{\sum_{j=1}^{N} p_y(j)(j - \mu_y)^2}$$

Probabilità

$$p_{x+y}(k) = \sum_{i=1}^{N}\sum_{j=1}^{N} p(i,j) \qquad \begin{matrix} i+j=k \\ k = 2, 3, \dots, 2N \end{matrix}$$

$$p_{x-y}(k) = \sum_{i=1}^{N}\sum_{j=1}^{N} p(i,j) \qquad \begin{matrix} |i-j|=k \\ k = 0, 1, \dots, N-1 \end{matrix}$$

Entropie

$$HX = -\sum_{i=1}^{N} p_x(i) \cdot \log p_x(i)$$

$$HY = -\sum_{j=1}^{N} p_y(j) \cdot \log p_y(j)$$

$$HXY1 = -\sum_{i=1}^{N}\sum_{j=1}^{N} p(i,j) \cdot \log\left[p_x(i) \cdot p_y(j)\right]$$

$$HXY2 = -\sum_{i=1}^{N}\sum_{j=1}^{N} p_x(i) \cdot p_y(j) \cdot \log\left[p_x(i) \cdot p_y(j)\right]$$

Coefficiente Q

$$Q(i,j) = \sum_{k=1}^{N} \frac{p(i,k) \cdot p(j,k)}{p_x(i) \cdot p_y(j)}$$

È ora possibile definire le 14 features di Haralick.

1) Energia $$f_1 = \sum_{i=1}^{N}\sum_{j=1}^{N} p(i,j)^2$$

L'energia misura l'ordine di un'immagine, ossia la regolarità della differenza tra due pixel dell'immagine stessa. La funzione utilizza ciascun valore $p(i,j)$ della matrice di co-occorrenza come peso, in quanto in un'immagine ordinata ciascuna coppia di valori si presenta un numero di volte maggiore rispetto al caso di un'immagine disordinata. Quindi, il peso aumenta all'aumentare dell'ordine e diminuisce all'aumentare del disordine. In Figura 8 è mostrato un esempio.

0	1	2	3	4
0	1	2	3	4
0	1	2	3	4
0	1	2	3	4
0	1	2	3	4

0	3	4	2	2
2	1	3	4	1
0	3	2	1	0
2	1	0	3	3
3	2	4	0	3

Figura 8. L'immagine a sinistra risulta più ordinata di quella a destra e la sua energia sarà dunque maggiore. Infatti, il peso è dato dalla matrice di co-occorrenza e quest'ultima, nel primo caso, risulta avere occorrenze maggiori: considerano la relazione spaziale con distanza 1 e angolo 0°, si nota che, nell' immagine di sinistra, l'1 è vicino al 2 cinque volte, il 3 è vicino al 4 cinque volte, ecc; nella seconda, invece, l'1 è di fianco al 2 zero volte, il 3 è di fianco al 4 due volte, ecc.

2) Contrasto $$f_2 = \sum_{k=1}^{N-1} k^2 \left[\sum_{i=1}^{N}\sum_{j=1}^{N} p(i,j)\right] \qquad |i-j| = k$$

Quantifica la differenza tra i toni di grigio di un pixel e del suo intorno. Quando *i* e *j* sono uguali, ossia i pixel considerati sono completamente simili al loro intorno, il peso k^2 nella funzione f_2 è pari a $k = |i-j| = 0$, quindi $f_2 = 0$, nessun contrasto. Se i e j differiscono di 1, vi è un piccolo contrasto e il peso k^2 vale 1. Se differiscono di 2, il contrasto aumenta e il peso corrispondente risulta essere $k^2 = 4$. Il peso k^2 aumenta esponenzialmente all'aumentare della differenza tra i toni di grigio di i e j.

3) Correlazione $$f_3 = \sum_{i=1}^{N}\sum_{j=1}^{N} \frac{(i-\mu_x)(i-\mu_y)p(i,j)}{\sigma_x\sigma_y}$$

La correlazione esprime la dipendenza lineare dei livelli di grigio tra pixel vicini. Un'alta correlazione tra due pixel significa un'alta prevedibilità della relazione, espressa attraverso l'equazione di regressione lineare, tra i

due pixel vicini. Solitamente i pixel sono più facilmente correlati ai pixel a loro vicini, rispetto ai pixel a loro lontani. Questa caratteristica può essere utilizzata per stimare le dimensioni di un oggetto all'interno dell'immagine analizzata. Nel codice scritto, si è posta la condizione di controllo su σ_x e σ_y: nel caso in cui una delle due sia zero, la correlazione viene posta uguale a 1.

4) Varianza $$f_4 = \sum_{i=1}^{N}\sum_{j=1}^{N}(i-\mu)^2 \cdot p(i,j)$$

La varianza è la misura della dispersione dei valori attorno al valore medio.

5) Omogeneità $$f_5 = \sum_{i=1}^{N}\sum_{j=1}^{N}\frac{p(i,j)}{1+(i-j)^2}$$

Al contrario del contrasto, l'omogeneità misura la similitudine tra i toni di grigio di un pixel e del suo intorno. In questo caso, infatti, se la differenza tra i toni di grigio di i e j diminuisce e quindi i due pixel considerati sono sempre più simili, l'omogeneità dell'immagine aumenta.

6) Somma delle medie $$f_6 = \sum_{k=2}^{2N} k \cdot p_{x+y}(k)$$

7) Somma delle varianze $$f_7 = \sum_{k=2}^{2N}(k-f_6)^2 \cdot p_{x+y}(k)$$

8) Somma delle entropie $$f_8 = -\sum_{k=2}^{2N} p_{x+y}(k) \cdot log\, p_{x+y}(k)$$

9) Entropia $$f_9 = -\sum_{i=1}^{N}\sum_{j=1}^{N} p(i,j) \cdot log\, p(i,j)$$

In questo contesto, l'entropia risulta l'opposto dell'energia e misura quindi il livello di disordine nell'immagine. Ciascun valore $p(i,j)$ della matrice di co-occorrenza è sempre compreso tra 0 e 1, in quanto la matrice è scritta sotto forma di probabilità. Il logaritmo di tale quantità sarà sempre 0 o negativo: minore il valore $p(i,j)$ (minore probabilità dell'occorrenza di quella specifica combinazione tra pixel), maggiore il valore assoluto di $log\, p(i,j)$, maggiore il valore dell'entropia. Il segno negativo davanti alla funzione rende positivo ciascun termine.

10) Differenza di varianze $$f_{10} = -\sum_{k=0}^{N-1}\left[k-\mu_{x-y}\right]^2 p_{x-y}(k)$$

11) Differenza di entropie $$f_{11} = -\sum_{k=0}^{N-1} p_{x-y}(k) \cdot log\, p_{x-y}(k)$$

12) Misura di correlazione 1 $$f_{12} = \frac{f_9 - HXY1}{max(HX, HY)}$$

Nel codice è stato posto un controllo su $max(HX, HY)$. Nel caso in cui tale massimo sia zero, la funzione f_{12} è posta pari a $f_9 - HXY1$.

13) Misura di correlazione 2 $$f_{13} = \sqrt{1 - e^{-2(HXY2 - f_9)}}$$

14) Coefficiente di massima correlazione $$f_{14} = \sqrt{second\ largest\ eignvalue\ of\ Q}$$

È opportuno sottolineare che nel codice creato, tutte le volte in cui è stato calcolato il logaritmo di una variabile *x* si è utilizzata la funzione *math.log(x+x==0)* che permette di trasformare ciascun $log(0)$ in $log(1)$ per non far perdere di significato la funzione matematica.

Il numero di livelli di grigio dell'immagine normalizzata determina le dimensioni della matrice di co-occorrenza e pertanto modifica il valore delle features di Haralick. Per questa correlazione, è importante scegliere in modo opportuno il numero di livelli di grigio attraverso i quali normalizzare l'immagine ed utilizzare sempre il medesimo numero per tutte le successive immagini importate, considerando il trade off tra una maggiore specificità e il tempo computazionale.

3.4 Modalità di calcolo sulle windows e visualizzazione

Si illustra ora il processo compiuto per l'estrazione delle 14 features dall'immagine di input e la successiva visualizzazione di queste sotto forma di immagini. Sono state utilizzate le stesse 6 slices ridimensionate (array di dimensioni [100, 100, 6] (vedi Figura 5), utilizzate per la visualizzazione della matrice di co-occorrenza in quanto, anche in questo caso, è una buona pratica selezionare un range di immagini dalle 275 importate o il tempo di computazione risulterebbe troppo elevato. È anzitutto necessario definire una piccola "finestra" su cui calcolare la matrice di co-occorrenza totale normalizzata da utilizzare per estrarre le features. Infatti, il processo di calcolo ed estrazione delle features da un'immagine è così svolto:

- si divide l'immagine di input in cubi (o quadrati, se l'immagine di input è bidimensionale) di dimensioni 5x5x5 (5x5x1);
- si calcola la matrice di co-occorrenza totale normalizzata di tale porzione dell'immagine;
- si computano le 14 features utilizzando la sopracitata matrice;
- si assegna al punto centrale della porzione un vettore con le 14 features come elementi;
- si ripetono i precedenti step per ogni pixel dell'immagine di input.

In particolare, sono state inizializzate le variabili *n_rows*, *n_cols* e *n_dep* con le tre dimensioni dell'immagine di input ridotta (nel nostro caso, *n_rows* = 512, *n_cols* = 512 e *n_dep* = 6) e il valore della variabile *windows_size*, che definisce la dimensione della finestra, è stato posto uguale a 5. Attraverso le seguenti righe di codice:

if n_dep == 1: windows_dep = 0

else: windows_dep = windows_size

è stato effettuato il controllo sulla dimensione di *n_dep*: nel caso in cui *n_dep* sia uguale a 1, e quindi l'immagine di input sia bidimensionale (composta da una sola slice), *windows_dep* è inizializzato a 0 e la

"finestra" avrà quindi solo dimensioni 5x5x1, altrimenti, la terza dimensione della "finestra", la sua profondità, sarà posta uguale alle altre due dimensioni, in questo caso caso 5.

Segue il ciclo per il calcolo delle 14 features:

```
for c in range (windows_size//2, n_cols-windows_size//2, 1):
    for r in range (windows_size//2, n_rows-windows_size//2, 1):
        for d in range (windows_dep//2, n_dep-windows_dep//2, 1):
            x = c - windows_size//2
            y = r - windows_size//2
            g = d - windows_dep//2
            w = x + windows_size
            h = y + windows_size
            l = g + windows_dep
            if l == 0: l = 1
            image_to_compute = image_reduced.copy()[y:h,x:w,g:l]
            features_matrix[c,r,d] = calculate_features(image_to_compute, levels)
```

Il primo pixel dell'immagine a ricevere l'array di features non potrà essere il pixel in posizione (0, 0, 0) in quanto la finestra 5x5x5 è, in tale posizione, impossibile da costruire. Dovrà invece avere coordinate (*windows_size//2, windows_size//2, windows_dep//2*), ossia (2, 2, 2). Così come l'ultimo pixel a ricevere l'array di features non sarà l'ultimo pixel dell'immagine (512, 512, 6), ma quello con coordinate (*n_cols-windows_size//2, n_rows-windows_size//2, n_dep-windows_dep//2*), cioè (510, 510, 4). Per questo motivo una cornice di pixel dell'immagine iniziale rimarrà inoccupata, come mostrato in Figura 9.

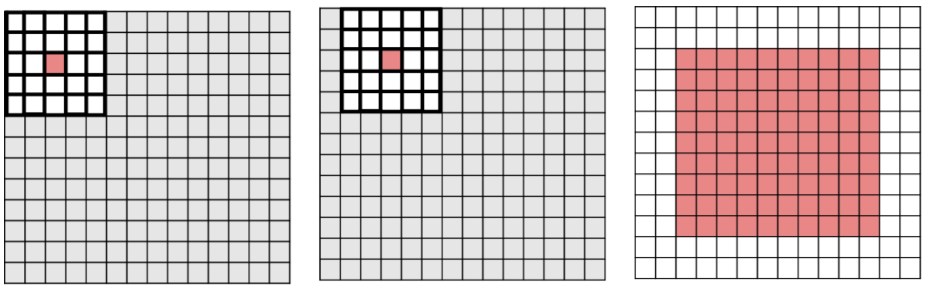

Figura 9. Schema rappresentante lo spostamento della "finestra" all'interno dell'immagine di input nel caso bidimensionale. L'immagine a sinistra mostra il primo pixel (colore rosso) che riceverà le 14 features computate a partire dalla matrice di co-occorrenza totale normalizzata, a sua volta calcolata sulla porzione di immagine 5x5 evidenziata in nero. Una volta assegnato al pixel centrale l'array con le 14 features, la "finestra" si sposta al pixel successivo e compie gli stessi steps (immagine centrale). L'area rossa nell'ultima immagine corrisponde all'insieme di pixel che hanno ricevuto le 14 features, la restante cornice di pixel bianchi non ha ricevuto le features.

Dunque, per ogni pixel di ciascuna colonna, riga e profondità nel range (2, 2, 2) – (510, 510, 4) dell'immagine di input viene assegnata alla variabile *image_to_compute* una copia della porzione 5x5x5 dell'immagine iniziale. La funzione *calculate_features()* riceve tale porzione come argomento, la utilizza per calcolare la matrice di co-occorrenza totale normalizzata e per estrarre le features. Quest'ultime sono assegnate al vettore *features_matrix* nel punto con le medesime coordinate del pixel selezionato.

Utilizzando un'immagine di dimensioni [512, 512, 6] il codice impiega 380.6 secondi per completare la computazione.

È ora possibile rappresentare graficamente le 14 features calcolate per ciascuna slice dell'immagine iniziale. Avendo scelto 6 slices ed avendo ottenuto una cornice di 2 pixel agli estremi di ciascuna dimensione, la porzione tridimensione di immagine per la quale sono state calcolate le features ha dimensioni [510, 510, 2]. Si ottengono quindi 14 immagini delle 14 features per ciascuna delle 2 slices di immagine.

La funzione *plt.subplots()* permette di creare una griglia suddivisa per celle all'interno delle quali vengono riportatetutte le immagini desiderate. La riga di codice

```
fig, axs = plt.subplots(n_dep-2*(windows_dep//2), 14, figsize = (50*(n_dep-2*(windows_dep//2)),15))
```

crea una griglia con *n_dep-2*(windows_dep//2)* = 2 righe e 14 colonne su cui plottare immagini di dimensioni 100x15 attraverso i comandi:

```
axs = axs.flatten()

ax = 0

for i in range (windows_dep//2, n_dep-(windows_dep//2), 1):

    for j in range (14):

        media = np.mean(features_matrix[2:-2,2:-2,i,j])

        features_matrix[:2,:,i,j] = media

        features_matrix[-2:,:,i,j] = media

        features_matrix[:,-2:,i,j] = media

        features_matrix[:,:2,i,j] = media

        im = features_matrix[:,:,i,j]

        axs[ax].set_title('Slice n°: %s    Features n°: %s' % (i, j+1))

        axs[ax].imshow(im)

        ax += 1.
```

All'interno del ciclo è assegnato alla variabile *media* il valor medio delle features, così da attribuire alla cornice esterna tale valor medio. Questa assegnazione viene effettuata per evitare che i valori nulli assegnati alla cornice possano influire sulla scala di colori delle immagini rappresentate.

La rappresentazione grafica delle features è riportata in Figura 10.

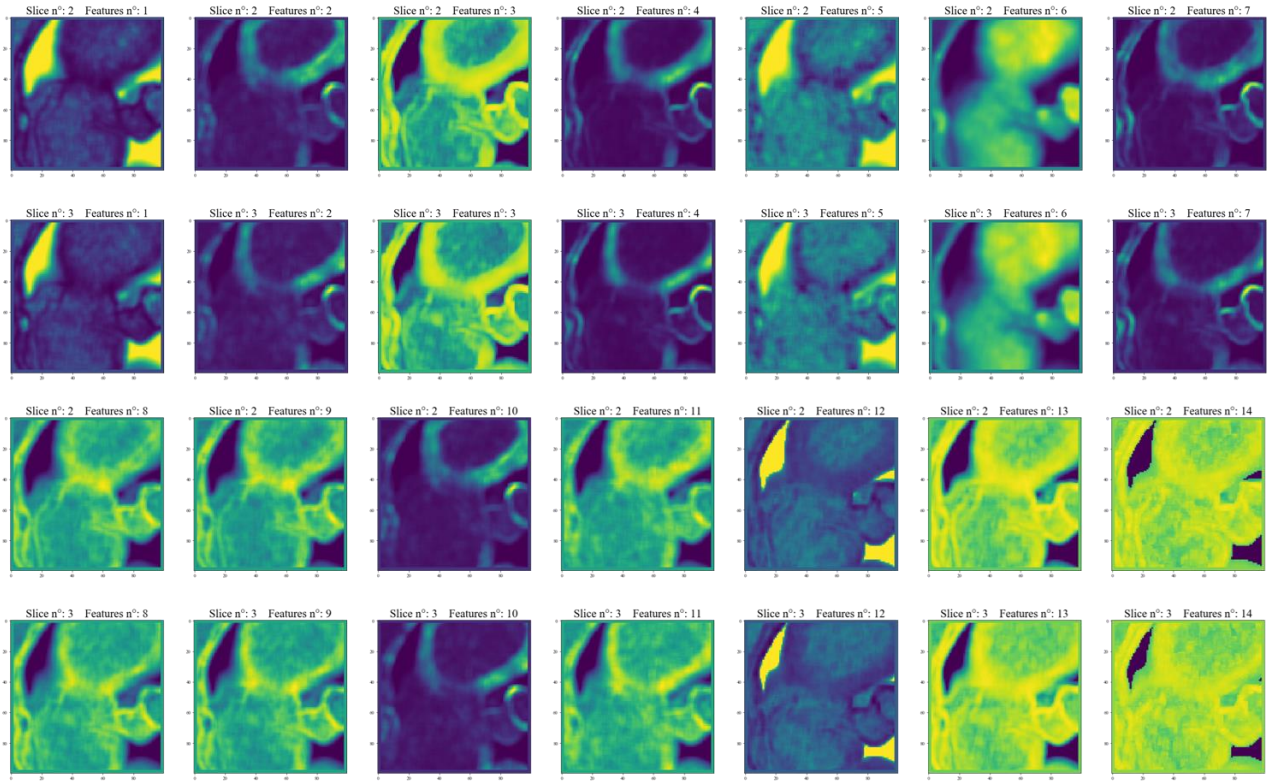

Figura 10. Rappresentazione grafica delle features estrapolate da un'immagine di dimensioni [512,512,6], con una cornice di 2 pixel per dimensione: solo le slices numero 2 e numero 3 saranno rappresentate. Le prime due righe di immagini contengono le prime 7 features delle due slices a confronto. Le immagini della terza e della quarta riga contengono le features dalla ottava alla quattordicesima per le due slices a confronto.

Capitolo 4. Modello per la classificazione dei tessuti

4.1 Confronto dei modelli testati per la classificazione dei tessuti

Gli algoritmi per il machine learning [4] possono essere classificati in tre grandi categorie a seconda del tipo di apprendimento:

- Supervisionato. Al modello vengono forniti esempi di possibili input e rispettivi output correlati e l'obiettivo è quello di estrarre una regola generale che associ a un input sconosciuto l'output corretto.
- Non supervisionato. Il modello ha lo scopo di trovare una struttura negli input forniti, senza che questi siano etichettati in alcun modo.
- Apprendimento per rinforzo. Il modello interagisce con un ambiente dinamico nel quale cerca di raggiungere un obiettivo, avendo un insegnante che gli dice solo se ha raggiunto l'obiettivo.
- Semi supervisionato. Nel quale l'insegnante fornisce un dataset incompleto per l'allenamento, cioè un insieme di dati per l'allenamento tra i quali ci sono dati senza il rispettivo output corrispondente.

Una diversa classificazione dei modelli del machine learning si ottiene considerando lo scopo del loro utilizzo:

- Classificazione. Gli output sono divisi in classi e il sistema di apprendimento deve produrre un modello che assegni gli input non ancora visti a una o più di queste. Tipicamente il modello è di tipo supervisionato.
- Regressione. I dati utilizzati in questo caso hanno valori continui e il modello deve prevedere il valore futuro di una determinata variabile conoscendo i valori acquisiti nel passato. Tipicamente il modello è di tipo supervisionato.
- Clustering. Lo scopo è la divisione in gruppi dei dati di input, ma diversamente da quanto accade per la classificazione, i gruppi non sono preventivamente noti. Risulta tipicamente di tipo non supervisionato.

Nel corso di questo capitolo ci si concentrerà unicamente sulla classificazione, in quanto lo scopo di questo elaborato è la classificazione dei diversi tessuti presenti in un'immagine medica.

L'allenamento di un modello per la classificazione è essenzialmente divisibile in due fasi: una fase di training e una fase di testing. Nella prima fase, il modello software utilizza un campione di dati all'interno del quale a ogni elemento è assegnata una label. Per ciascun elemento vengono determinati i descrittori caratteristici della classe a cui appartiene. Nella seconda fase, gli stessi descrittori sono calcolati su elementi ignoti (a cui non è stata assegnata una label) e sono poi comparati con quelli precedentemente ottenuti con lo scopo di assegnare la label della classe che meglio si adatta.

Gli algoritmi supervisionati utilizzati in questo caso per la classificazione dei tessuti sono: Decision Tree, Random Forest, XGBoost, SVM Classifier, K-Nearest Neighbors Classifier, SGD Classifier e Gaussian Naive Bayes Classifier.

I modelli prodotti dalle reti neurali artificiali sono modelli computazionali composti di "neuroni" artificiali, ispirati dalla semplificazione di una rete neurale biologica. A differenza di un sistema algoritmico, dove si può esaminare passo-passo il percorso che dall'input genera l'output, una rete neurale è una "black box" in grado di generare un risultato valido (con una alta probabilità di essere accettabile) senza che sia possibile spiegare come tale risultato sia stato generato. I modelli di reti neurali utilizzati per la classificazione dei tessuti sono: MLP Neural Network, Logistic Regression e Linear Discriminant Analysis.

Si presenterà ora una breve spiegazione semplificata del funzionamento dei modelli e degli algoritmi utilizzati.

1) Decision Tree

Il Decision Tree [5] ha una struttura simile a un diagramma di flusso. È costituito da nodi interni (il primo in alto è detto nodo radice, i successivi sono detti nodi figlio) che rappresentano i sottoinsiemi del dataset inziale divisi a seconda degli attributi; rami, che rappresentano le regole decisionali secondo cui avviene la divisione; nodi foglia, che rappresentano i sottoinsiemi finali in cui il database iniziale è stato suddiviso. L'algoritmo seleziona, per ciascun nodo interno, l'attributo migliore utilizzando l'Attribute Selection Measures (ASM) ed impara a partizionare i dati di input in modo ricorsivo in base al valore di tale attributo. L'ASM definisce il criterio di divisione per separare i dati nel miglior sistema possibile, facendo cioè in modo che gruppi diversi siano il più differenti possibile l'uno dall'altro e che i membri di ogni gruppo risultino il più simili possibile l'uno dall'altro. Una volta scelto l'attributo migliore, l'algoritmo lo imposta come nodo decisionale e suddivide così il dataset inziale in subsets più piccoli, ripete in modo ricorsivo gli step appena descritti per ciascun elemento figlio. Un esempio è riportato in Figura 11. Il processo si arresta quando tutte le istanze del subset appartengono allo stesso valore di attributo, o non ci sono più attributi, o non ci sono più istanze.

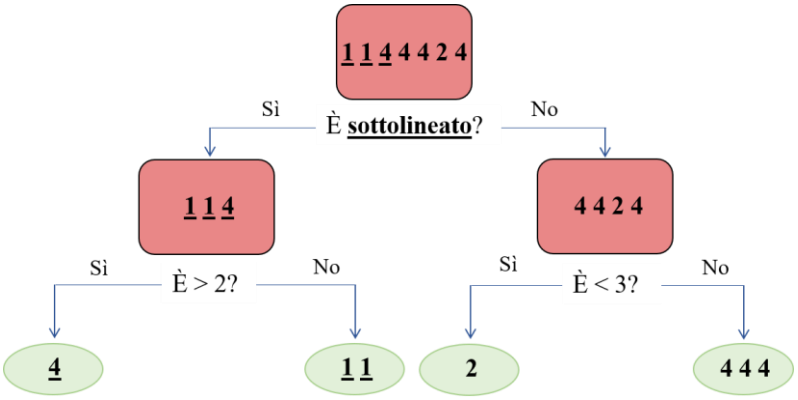

Figura 11. Esempio schematico della struttura decisionale di un digramma ad albero. I rettangoli rossi costituiscono i nodi interni, di cui primo in alto è il nodo radice e i successivi sono i nodi figlio, le linee blu sono le regole decisionali e gli ovali verdi indicano i nodi foglia. In base al valore dell'attributo selezionato come migliore, le regole decisionali dividono il dataset iniziale in sottoinsiemi fino a raggiungere il risultato di classificazione finale.

2) Random Forest

La Random Forest [6] è un algoritmo costituito da un gran numero di Decision Trees che operano insieme in modo parallelo (*bagging*, vedi Figura 12): ciascun albero si comporta come un weak learner, un classificatore la cui precisione sul risultato è leggermente maggiore del 50%, ed è quindi di poco migliore della scelta a caso, e dà come risultato la previsione di una classe; la classe con il maggior numero di occorrenze diventa la previsione del modello (vedi la Figura 13).

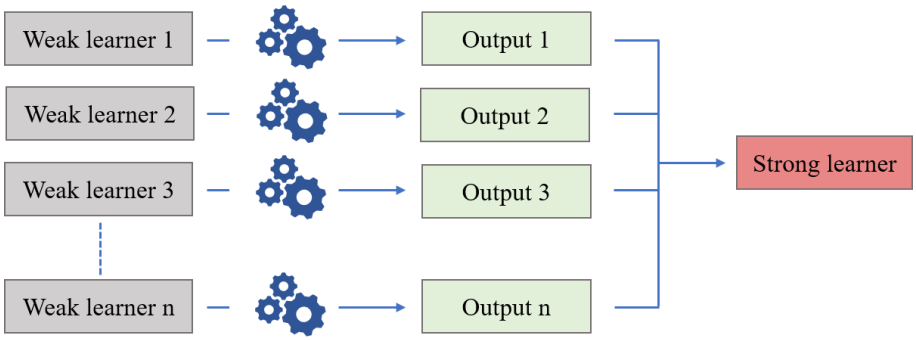

Figura 12. Schema di un algoritmo di tipo bagging. L'algoritmo "strong learner" è costituito da vari "weak learner" che operano in parallelo e forniscono un risultato ciascuno. Tali risultati sono utilizzati per ottenere il risultato del modello.

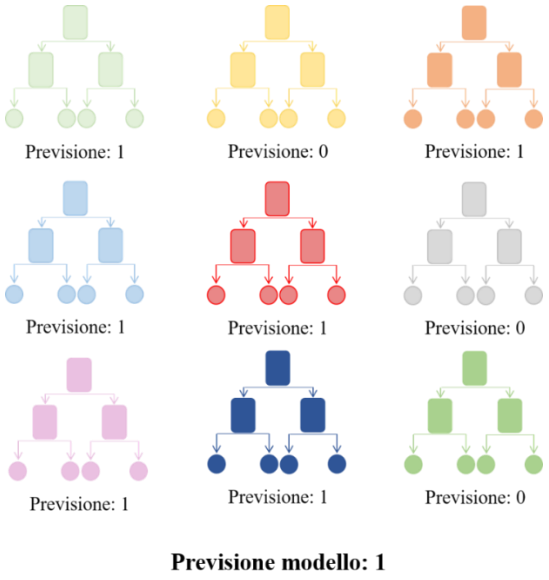

Figura 13. Schema decisionale della Random Forest: ciascun Decision Tree fornisce come risultato la previsione della classe, la previsione con il maggior numero di voti costituirà la previsione finale del modello.

Il motivo per cui questo modello funziona così bene è che un gran numero di modelli relativamente non correlati (i singoli Decision Tree da cui è composto il modello) che operano insieme produce una previsione più accurata di uno qualsiasi dei singoli modelli costituenti. La bassa correlazione tra i singoli modelli costituenti è la chiave del successo di questo algoritmo: i singoli alberi si "proteggono" a vicenda dagli errori individuali (a condizione che non sbaglino contemporaneamente nella stessa direzione) e, mentre alcuni alberi potrebbero dare risultati sbagliati, i molti altri alberi che forniranno un risultato corretto faranno in modo di spostare il risultato nella direzione corretta.
Inoltre, ogni Decision Tree della Random Forest considera un sottoinsieme casuale di attributi nel formare le regole decisionali e ha accesso solo a un set casuale dei dati di training. Questo aumenta la diversità nella foresta e porta a previsioni generali più solide.

3) XGBoost

XGBoost (Extreme Gradient Boosting) [7] è la versione potenziata del Gradient Boosting e fa quindi parte della famiglia dei *boosting*. L'idea che sta alla base di questa tipologia di algoritmi è quella di costruire una sequenza di "weak learner" (vedi Figura 14), la cui precisione sul risultato è leggermente maggiore del 50%, ed è quindi di poco migliore della scelta a caso e, a ogni iterazione, assegnare a ciascun risultato mal classificato nell'iterazione precedente un peso grande e procedere a una nuova classificazione. Questi step vengono ripetuti e, iterazione dopo iterazione, la classificazione sarà sempre più accurata. Il processo continua fino a quando l'algoritmo è in grado di classificare l'output in modo corretto. È quindi stato costruito uno "strong learner" che sfrutta l'errore di classificazione dell'iterazione precedente per ridurre l'errore nella classificazione successiva, vedi Figura 15.

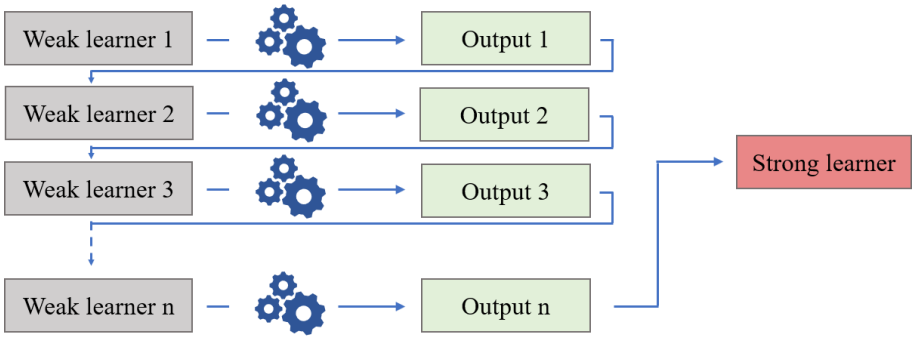

Figura 14. Schema di un algoritmo di tipo boosting. I vari "weak learner" operano in sequenza per migliorare ad ogni nuova iterazione la classificazione, fino a diventare "strong learner".

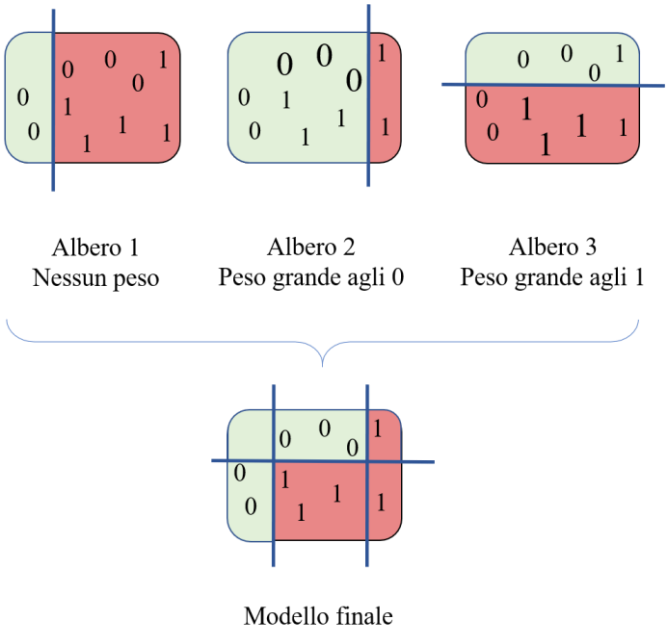

Figura 15. Esempio schematico del funzionamento dell'algoritmo XGBoost. Il primo albero divide gli elementi con una linea verticale: tutto ciò che è a sinistra di tale linea è uno 0, tutto ciò che si trova a destra è un 1. Alla seconda iterazione, il secondo albero assegna ai tre zeri che il primo albero aveva erroneamente classificato come uni un peso grande e classifica nuovamente con una linea verticale: tutto ciò che è a sinistra è uno 0, tutto ciò che si trova a destra è un 1. Il terzo albero, alla terza iterazione, assegna un peso grande ai tre uni che l'albero precedente aveva classificato in modo non esatto e tenta una nuova classificazione con una linea orizzontale: tutto ciò che si trova sopra tale linea è uno 0, tutto ciò che si trova sotto è un 1. Il modello finale utilizza una combinazione pesata dei primi tre "classificatori deboli" trasformandosi in un "modello forte" in grado di classificare tutti i numeri senza errori.

Nel caso specifico dell'XGBoost i vari "weak learner" sono alberi decisionali con un'unica divisione (ceppi decisionali) e, a ogni iterazione, il peso assegnato agli output non corretti si basa sull'ottimizzazione (riduzione) della funzione di perdita dell'iterazione precedente.

Alla fine del processo i punteggi di stima di ogni singolo albero vengono combinati per ottenere il punteggio finale (vedi Figura 16, che riprende un esempio tratto dalla documentazione online di XGBoost per Python [8]). La Figura 16 mostra due diagrammi ad albero (ceppi decisionali), ciascuno dei quali propone una diversa classificazione di un gruppo di persone in base all'utilizzo dei videogiochi. Il primo ceppo decisionale divide le persone in base all'età, il secondo in base alla frequenza di utilizzo del computer. Al termine del processo, il punteggio di stima dell'apprezzamento dei videogiochi è una combinazione dei pesi dei singoli alberi decisionali.

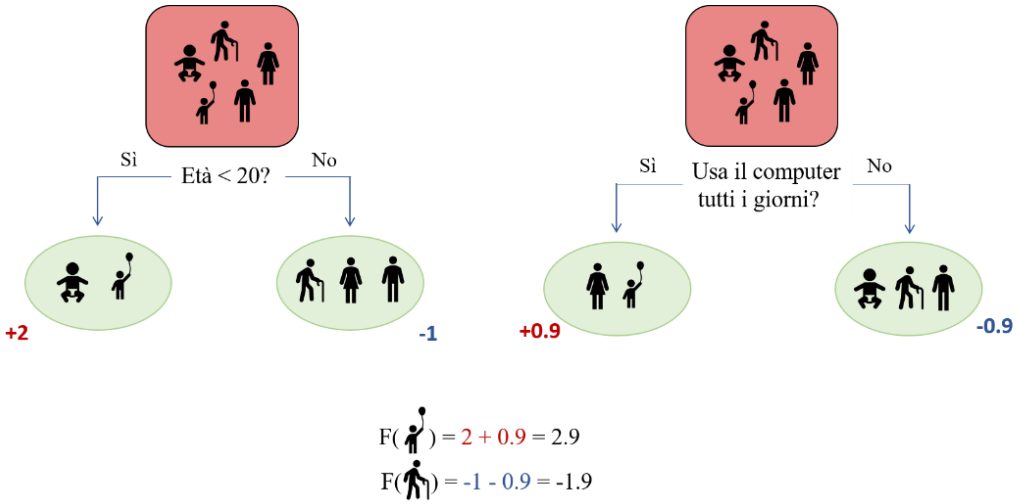

Figura 16. Esempio di classificazione di un gruppo di persone in base all'utilizzo dei videogiochi. Il primo albero propone una classificazione in base all'età

4) SVM Classifier

Il Support Vector Machine (SVM) [9] è un classificatore che permette di separare lo spazio delle classi mediante un iperpiano che massimizzi la distanza tra esse: rappresentato graficamente il dataset di input in un iperspazio, l'algoritmo restituisce l'iperpiano ottimale per dividere le classi presenti nell'iperspazio. Un esempio del caso bidimensionale è presentato in Figura 17.

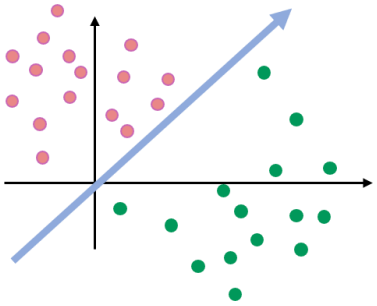

Figura 17. Esempio della creazione dell'iperpiano nel caso bidimensionale (retta) per la classificazione dei punti verdi da quelli rossi.

Nelle applicazioni reali difficilmente i dati risulteranno così perfettamente divisibili nello spazio. In Figura 17è illustrato un semplice esempio di come si potrebbe ipoteticamente comportare l'algoritmo nel caso in cui i dati appaiano parzialmente sovrapposti. Entrambe le metodologie risultano corrette: la prima tollera alcuni punti anomali nella classificazione, la seconda ha tolleranza zero e raggiunge la partizione perfetta. Nei casi reali è necessario considerare il trade off tra l'accuratezza della classificazione e il tempo impiegato per raggiungerla. Infatti, trovare una classificazione perfetta per milioni di set di dati richiede molto tempo computazionale.

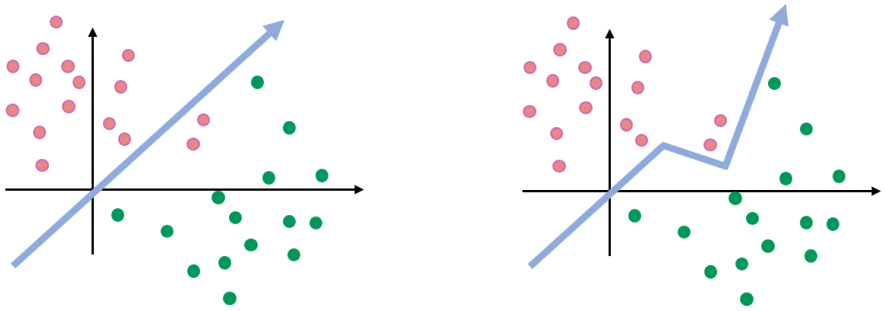

Figura 18. Esempio della costruzione della retta per la classificazione dei dati nel caso in cui siano parzialmente sovrapposti nello spazio.

Inoltre, la funzione "Kernel-trick" permette di classificare anche i dati non-lineari; tale funzione è sempre presente nell'algoritmo SVM ma entra in gioco solo quando i dati risultano inseparabili con tecniche lineari.

5) K-Nearest Neighbors Classifier

Il K-Nearest Neighbors Classifier [10] è un algoritmo in cui l'oggetto da classificare viene rappresentato graficamente in uno spazio tridimensionale in base agli attributi che possiede e viene poi classificato in base al suo intorno, in particolare gli viene assegnata la classe a cui appartiene la maggioranza dei K campioni più vicini. In Figura 19 è rappresentato un dataset formato da punti blu, verdi e rossi. Attraverso il K-Nearest Neighbors Classifier è possibile classificare il punto stella: impostando il valore di K a 3, il punto stella risulta essere verde.

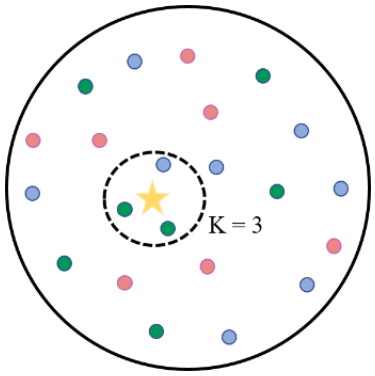

Figura 19. Schema bidimensionale di funzionamento dell'algoritmo di K-Nearest Neighbors. Il database inziale è formato da punti rossi, verdi e blu e l'obiettivo è quello di classificare il punto stella. Scegliendo K=3 il punto stella sarà classificato verde.

6) SGD Classifier

Lo Stochastic Gradient Descent [11] è un semplice ed efficiente algoritmo per la classificazione supervisionata che utilizza una funzione di perdita lineare e convessa. Supporta la classificazione multiclasse combinando più classificatori binari in uno schema OVA, One Versus All: l'algoritmo binario impara a distinguere ciascuna classe *k* dalle altre *k-1* classi. Nel momento del test, l'algoritmo calcola il "confidence score" per ciascuna classe a cui è stata assegnata una label e assegna al nuovo elemento da classificare la classe con il più alto livello di confidenza. In Figura 20 è illustrato l'approccio OVA per un dataset composto da tre diverse classi; i colori dello sfondo mostrano la superficie decisionale indotta da ciascuno dei tre classificatori.

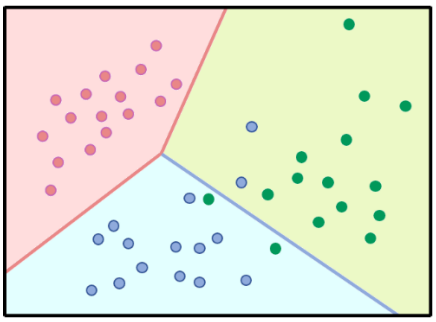

Figura 20. Schema della classificazione di un dataset composto da tre classi: pallini blu, pallini verdi e pallini rossi. I colori dello sfondo mostrano la superficie decisionale indotta da ciascuno dei tre classificatori.

7) Gaussian Naive Bayes Classifier

Il Naive Bayes Classifier [12] è un algoritmo della famiglia dei "classificatori probabilistici" basato sull'applicazione del teorema di Bayes e sull'ipotesi di forte indipendenza tra le variabili prese in esame. Ad esempio, volendolo applicare alla classificazione di un frutto, saranno prese in considerazione variabili come il colore, la forma, le dimensioni. Un frutto è classificato "mela" se è rosso, rotondo e di circa 8 cm di diametro. Il Naive Bayes Classifier considera che ognuna di queste features contribuisca in modo indipendente alla probabilità che il frutto sia una mela, senza tener presente le eventuali correlazioni tra il colore, la rotondità e il diametro.

Utilizzando il teorema di Bayes, la probabilità condizionale può essere scritta come

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)},$$

dove $P(A|B)$ è la probabilità a posteriori (*posterior*), ovvero la probabilità che avvenga l'evento A sapendo che è avvenuto l'evento B; $P(B|A)$ è la densità di probabilità condizionata (*likelihood*) che avvenga l'evento B sapendo che è avvenuto l'evento A; e $P(A)$ e $P(B)$ sono la probabilità a priori dell'evento A (*prior*) e la densità di probabilità assoluta dell'evento B (*evidence*), rispettivamente.

L'algoritmo calcola le probabilità per ogni fattore utilizzando la formula precedente e seleziona il risultato con la probabilità più alta (regola decisionale MAP, Maximum a Posteriori Probability). Mentre la stima delle probabilità a priori è abbastanza semplice (se non si hanno elementi si possono ipotizzare le classi equiprobabili), la conoscenza delle densità condizionali è possibile "solo in teoria". Spesso si fanno ipotesi sulla forma delle distribuzioni (es. distribuzione multinormale, Gaussiana, di Bernoulli) e si apprendono i parametri fondamentali dal training set. Nel Gaussian Naive Bayes Classifier, si assume che la probabilità delle features sia Gaussiana:

$$P(B_i|A) = \frac{1}{\sqrt{2\pi\sigma_A^2}} e^{-\frac{(B_i-\mu_A)^2}{2\sigma_A^2}}$$

dove μ è il valor medio e σ la deviazione standard.

8) MLP Neural Network

Il percettrone è un classificatore lineare costruito a somiglianza del neurone biologico. Come tale, svolge il compito che il neurone biologico svolge nelle reti neurali biologiche ed è definito da tre elementi schematizzati in Figura 21:

- sinapsi e dendriti: tradotti nei pesi w che insieme ai segnali di input x e al bias (o soglia di attivazione) b_k costituiscono l'input del neurone;
- nucleo: composto dal sommatore Σ e dalla funzione di attivazione (o trasferimento) φ. Il primo determina i contributi degli input e pertanto somma ciascun input x con il rispettivo peso w e aggiunge alla somma così ottenuta il bias b, $\sum_i (x_i \cdot w_i) + b_k$. La seconda è una funzione lineare applicata al risultato della funzione precedente e determina il comportamento del neurone, ovvero la sua attivazione o disattivazione in base al superamento o meno del valore di soglia.
- assone: interpretato come il segnale di uscita del neurone

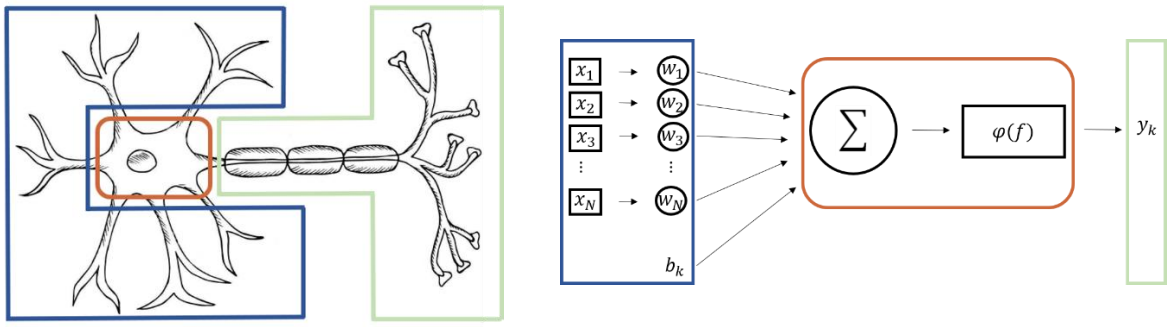

Figura 21. A sinistra, schema di un neurone biologico in cui sono evidenziati i dendriti e le sinapsi (in blu), il soma con il nucleo (in rosso) e l'assone (in verde). A destra, schema di un percettrone lineare k costituto da segnali di input e pesi sinaptici (in blu), funzione Σ e funzione di attivazione φ che determinano l'attivazione o la disattivazione del neurone (in rosso) e il segnale di output (in verde).

Il modello MLP Neural Network [13] consiste in almeno tre strati (layers) di percettroni: un input layer, un layer nascosto e un output layer (vedi Figura 22).

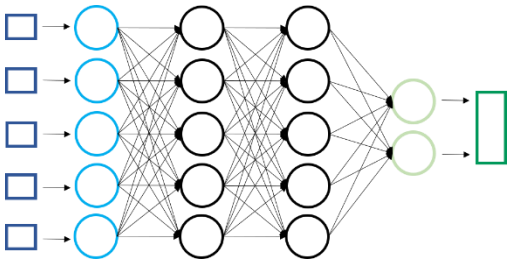

Figura 22. Esempio schematico di una rete neurale MLP costituita dai segnali di input (blu), un input layer (azzurro), due layer nascosti (in nero), un layer di output (verde chiaro), l'output della rete (verde scuro).

A differenza dei percettroni, nella MLP Neural Network la funzione di attivazione non è lineare. Di conseguenza il neurone non è acceso/spento ma la sua attività varia di intensità in base al suo livello di eccitazione. Ciò permette di ottenere un mapping più complesso dell'informazione di input. Una delle funzioni di attivazione maggiormente utilizzata è la sigmoide.

Fissata la topologia della rete neurale (numero di livelli e neuroni), l'addestramento consiste nel determinare il valore dei pesi che individuano il mapping desiderato tra input e output.

L'apprendimento avviene secondo la tecnica supervisionata di forward e backpropagation che si basa sulla modifica dei pesi di connessione in base all'errore nell'output rispetto al risultato previsto. Per evitare problemi di overfitting è opportuno modificare ad ogni iterazione le connessioni tra i vari layer così da impedire alla rete di "imparare a memoria" i collegamenti e obbligarla a costruire sempre nuove connessioni.

9) Logistic Regression

L'algoritmo di Logistic Regression è un algoritmo di Linear Regression nel quale invece di una funzione lineare viene utilizzata una funzione detta "logistic function" (o sigmoide) con valori compresi tra 0 e 1. Tale funzione converte il valore di input in un altro valore compreso tra 0 e 1, che costituisce la stima di probabilità. La classificazione si basa sulla scelta di un valore di soglia, compreso tra 0 e 1, al di sotto o al di sopra del quale l'input sarà classificato in classi diverse. In Figura 23 è riportato un esempio di classificazione [14] in cui il dataset di input è formato da elementi appartenenti alle classi "cane" e "gatto". Scegliendo come valore di soglia 0.5, tutti gli input mappati con valori maggiori di 0.5 saranno classificati "cane" e tutti gli input mappati con valori minori di 0.5 apparterranno alla classe "gatto".

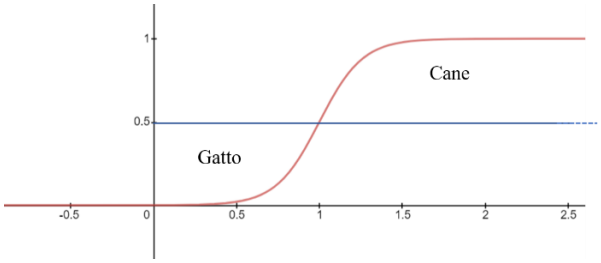

Figura 23. Esemplificazione della classificazione binaria nelle classi "cane" e "gatto" attraverso l'utilizzo del modello di Logistic Regression. È stato impostato il valore di soglia a 0.5 e pertanto, tutti gli input con valore maggiore sono classificati "cane", tutti gli input con valore minore sono classificati "gatto".

10) Linear Discriminant Analysis

Il modello di Linear Discriminant Analysis [15] è comunemente utilizzato come tecnica supervisionata di riduzione della dimensionalità di un set di dati da classificare. L'obiettivo è quello di proiettare lo spazio n-dimensionale delle features dei dati di input in un sottospazio di dimensione minore, rimuovendo le features ridondanti e dipendenti. La classificazione si basa sue tre step: calcolare la separabilità tra le classi differenti (ad esempio la distanza tra le medie di diverse classi), calcolare la separabilità tra gli elementi di una stessa classe, costruire uno spazio a dimensioni minori rispetto a quello di partenza in cui è massimizzata la separazione tra le diverse classi e minimizzata la separazione tra elementi di una stessa classe. In generale, così facendo, diminuisce il costo computazionale della classificazione e aiuta a evitare l'overfitting.

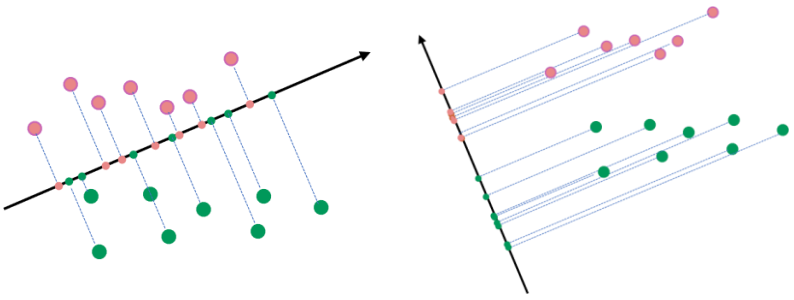

Figura 24. Schema di funzionamento dell'algoritmo di LDA in cui il sottospazio su cui sono proiettati i dati di input è una retta. Si nota che nell'immagine di destra è massimizzata la distanza tra classi diverse (pallini verdi e rossi) ed è minimizzata la distanza tra gli elementi di una stessa classe (pallini rossi tra loro e pallini verdi tra loro). In questo caso la classificazione avviene in modo corretto.

4.2 Accuracy e performances

Sono state analizzate l'accuracy sul training, l'accuracy sul test e i tempi computazionali dei vari modelli presi in esame poc'anzi. L'accuratezza della classificazione sul traing e sul test è la percentuale di pattern correttamente classificati, il tempo computazionale corrisponde invece al tempo che occorre per allenare il modello. Ci si è basati su tali parametri, riportati in Tabella 1, per scegliere l'algoritmo migliore da utilizzare per la classificazione dei diversi tessuti presenti nelle immagini DCOM: è stato scelto l'XGBoost Classifier.

	Decision Tree	Random Forest	XGBoost	SVM	K-Nearest Neighbors	SGD	Gaussian Naive Bayes	MPL	Logistic Regression	LDA
Training time [s]	0.049	0.053	1.179	0.094	0.008	0.062	0.0	3.206	0.313	0.016
Accuracy on train	91.33%	99.66%	95.76%	92.09%	94.27%	81.07%	82.26%	92.12%	88.95%	88.36%
Accuracy on test	89.72%	93.50%	92.54%	91.41%	91.53%	80.90%	81.81%	91.86%	89.38%	89.04%

Tabella 1. Sono mostrati i tempi di computazione, l'accuratezza della classificazione sul tain e sul test per ciascun modello utilizzato.

4.3 Creazione dei dataset per la classificazione dei tessuti

Prima di poter procedere con l'allenamento del modello scelto, è necessario costruire i dataset delle label e delle features. Si è scelto di costruire due dataset differenti per le label e per le features cosicché, nel caso in cui si decida di modificare la dimensione della finestra (variabile *size* all'interno del codice) utilizzata per l'estrazione delle features, non è necessario eseguire nuovamente il labelling del database, in quanto le coordinate dei punti nell'immagine, salvati nel dataset delle label, rimarrebbero in ogni caso gli stessi. Si procede innanzitutto con la determinazione di quest'ultimo database. Lo scopo è quello di registrare una serie di punti, caratterizzati dalle coordinate (*x, y*) della slice dell'immagine a cui appartengono e dalla slice stessa, assegnando a ciascuno di essi la label "bone" (ossa), "organ" (organo) o "background" (sfondo).

Viene quindi scelta e visualizzata una slice, *slice_number*, dell'immagine *image* importata, definito un array chiamato *array_cropping* e inizializzate le variabili *size* e *step*, rispettivamente a 5 e 3. Attraverso la linea di codice

rs = widgets.RectangleSelector(ax, onclick, drawtype = 'box')

è richiamata la funzione *onclick()* definita nel seguente modo:

```
def onclick(eclick, erelease):
    if eclick.xdata + size < erelease.xdata:
        for i in range(int(eclick.xdata)+size//2,int(erelease.xdata)-size//2,step):
            for j in range(int(eclick.ydata)+size//2,int(erelease.ydata)-size//2,step):
                x, y = i-size//2, j-size//2
                ax.scatter(i,j,color = 'r', s = 1)
                array_cropping.append([i, j])
        ax.add_patch(Rectangle((eclick.xdata-size//2,eclick.ydata-size//2),erelease.xdata-eclick.xdata,
                                erelease.ydata-eclick.ydata, alpha = 0.1, facecolor = 'b'))
```

```
        fig.canvas.draw()
    else:
        x, y = int(eclick.xdata), int(eclick.ydata)
        ax.scatter(eclick.xdata, eclick.ydata, color = 'r', s = 1)
        ax.add_patch(Rectangle((x-size//2, y-size//2), size, size, alpha = 0.1, facecolor = 'b'))
        array_cropping.append([x, y])
        fig.canvas.draw()
```

La funzione permette di cliccare sull'immagine visualizzata per selezionare una serie di punti a cui assegnare una label. Riceve come argomenti eclick e erelase che corrispondono alle coordinate (*x, y*) rispettivamente del punto in cui si clicca il tasto sinistro del mouse e del punto in cui il tasto del mouse viene rilasciato. Questo permette di selezionare, oltre a un unico punto, anche un'area (un insieme di punti) dell'immagine. In quest'ultimo caso, che si verifica se *eclick.xdata + size < erelease.xdata*, cioè quando la coordinata *x* del punto in cui si clicca e la coordinata *x* del punto in cui si rilascia distano tra loro di più della lunghezza di *size*, ad ogni pixel ogni tre (numero definito da *step*) all'interno della regione individuata dal trascinamento del mouse viene assegnato un puntino rosso e le coordinate di ognuno di tali pixel vengono salvate in *array_cropping* attraverso *append()*. Altrimenti, se l'area individuata dal trascinamento del mouse risulta essere di dimensioni minori rispetto a (*size* x *size*), sono le coordinate (*x, y*) del punto in cui si clicca a essere evidenziate in rosso e salvate in *array_cropping* attraverso *append()*. Un esempio è riportato in Figura 25.

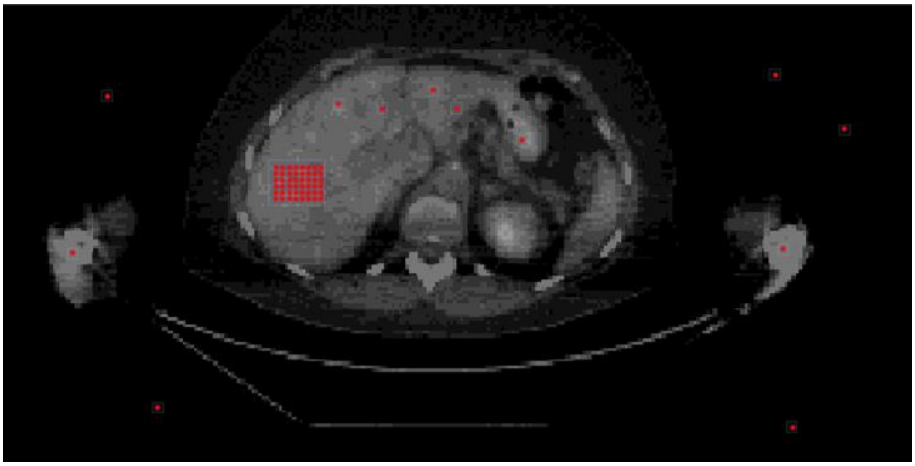

Figura 25. Immagine utilizzata per il labelling. Esempio di selezione di alcuni punti singoli e di un'area in cui sono salvati i più punti contenuti con passo pari a step impostato al valore 3.

È importante selezionare punti appartenenti tutti alla stessa label, ad esempio tutti punti appartenenti a organi.

A questo punto, è richiamata la funzione *save_array()*, riportata di seguito, che apre il file csv "labels.csv" e scrive il percorso *dir_path* dell'immagine, il numero della slice selezionata, la label assegnata ai punti selezionati nell'immagine e il contenuto di *array_cropping*, ossia le coordinate (*x, y*) dei punti selezionati.

```
def save_array(dir_path, array, label, slice_number):
    with open('labels.csv', 'a+') as f:
        for crop_points in array:
```

f.write('%s;%s;%s;%s;%s\n' % (dir_path, slice_number, label, crop_points[0], crop_points[1]))

f.close()

Il procedimento appena illustrato viene ripetuto per diverse slices dell'immagine e per tutte e tre le label, fino a quando non si sono ottenuti un numero sufficienti di dati per poter allenare il modello.

Successivamente, viene creato il database delle features. Il codice apre e legge il file "labels.csv" ed estrae le informazioni necessarie per costruire la finestra di dimensioni 5x5x5 su cui calcolare la matrice di co-occorrenza ed estrarre le features di Haralick, assegna quindi alle variabili x, y, g, w, h, l i vertici del cubo che ha come centro le coordinate dei punti precedentemente classificati:

x = row['x'] - size//2

y = row['y'] - size//2

g = row['slices'] - size//2

w = x + size

h = y + size

l = g + size

In particolare, *row['slices']* corrisponde al numero della slice dell'immagine selezionata, interpretata quindi come la profondità del cubo. Quindi, alla funzione *calculate_features()* è ora passata come argomento l'immagine ritagliata *image[y:h,x:w,g:l]*. Vengono poi creati due array, *array_features* e *array_labels*, riempiti rispettivamente con le 14 features estratte e la label ad esse associata. Attraverso le righe di codice:

feature_columns = ['f1','f2','f3','f4','f5','f6','f7','f8','f9','f10','f11','f12','f13','f14']

df_array_features = pd.DataFrame(array_features, columns = feature_columns)

df_array_features['array_labels'] = np.asarray(array_labels) #aggiungo gli array_labels

df_array_features.to_csv('features.csv', header = True, sep = ';', decimal = ',', encoding = 'Latin1', index = None)

df_train_data = pd.read_csv('features.csv', header = 0, sep = ';', decimal = ',', encoding = 'Latin1')

df_train_data = df_train_data.dropna()

features = df_train_data[feature_columns]

labels = df_train_data['array_labels']

sono eseguite le seguenti azioni: viene convertito l'array *array_features* in DataFrame utilizzando come intestazione delle colonne *feature_columns* (una colonna per features); viene aggiunta una quindicesima colonna con la label contenuta in *array_labels*; il DataFrame così composto viene convertito nel file csv "features.csv"; il file viene letto per eliminare tutte le righe contenenti valori delle features di tipo NaN (not a number); sono creati due DataFrame, *features* e *labels*, contenenti rispettivamente tutte le features non NaN e le label corrispondenti.

Prima di procedere con l'allenamento è opportuno bilanciare il numero di elementi per ogni classe per limitare gli squilibri. Infatti, è stato creato un dataset complessivo di 6909 elementi: 2639 classificati "background", 1776 elementi classificati "bone" e 2494 elementi classificati "organ". Con il seguente comando:

portion = df_train_data.groupby('array_labels').head(1770)

portion['array_labels'].value_counts()

si ottengono 1770 elementi classificati “bone”, 1770 elementi classificati “background” e 1770 elementi classificati “organ”. Infine, si sovrascrivono i DataFrame *features* e *label* son i nuovi valori modificati:

features = portion[feature_columns]

labels = portion['array_labels'].

4.4 Allenamento e testing del modello migliore

Nei modelli di machine learning [16] è sempre opportuno convalidare la stabilità del modello scelto per l’allenamento, in quanto non è scontato che un modello allenato sui dati di training si adatti con precisione a dati reali mai visti prima. A tale scopo, si utilizza la tecnica di cross-validation (convalida incrociata) in cui si suddivide il dataset in due porzioni complementari: la prima (traning set) è utilizzata per l’allenamento del modello e la seconda (testing set) per un test preliminare del modello stesso. L’obiettivo della cross-validation è dunque quello di verificare la capacità del modello di generalizzare l’apprendimento sviluppato alla classificazione di dati sconosciuti (su cui il modello non si è mai allenato). È possibile, in questo modo, limitare i problemi di overfitting che si verificano quando l’accuracy sul train è molto elevata e quella sul test risulta bassa, risultato di un modello che si è eccessivamente specializzato nella classificazione di elementi noti e non è in grado di prevedere la giusta classificazione per dati diversi da quelli che ha utilizzato per l’allenamento. In Tabella 1 sono mostrate le accuracy sul train e sul test dei modelli testati.

Per poter eseguire l’allenamento, quindi, si suddivide ciascuno dei due dataset, *features* e *label*, in due parti: *train_feature* e *train_labels* contenenti i primi due terzi dei dataset iniziali e *test_feature* e *test_labels* contenenti il terzo rimanente dei dataset iniziali. Si trasformano poi *train_feature* e *test_feature* in array.

*train_feature = features[:len(features)*2//3]*

*test_feature = features[len(features)*2//3:]*

*train_labels = labels[:len(features)*2//3]*

*test_labels = labels[len(features)*2//3:]*

train_feature = np.asarray(train_feature)

test_feature = np.asarray(test_feature)

Il dataset iniziale consiste però in blocchi successivi di punti con la stessa label. Prima di suddividere il dataset è quindi necessario procedere con lo shuffle (“mescolamento”) del dataset per modificare l’ordine dei punti acquisiti con le righe di codice:

x_sparse = coo_matrix(features)

features, x_sparse, labels = shuffle(features, x_sparse, labels, random_state = 0).

Infine, si esegue la funzione *run_model()* che acquisisce il modello scelto, lo allena attraverso la funzione *model.fit()* e stampa in output il tempo necessario per l’allenamento, l’accuracy sul train e l’accuracy sul test.

def run_model(model, alg_name):

t1 = time.time()

model.fit(train_feature, train_labels)

```
t2 = time.time()
print ('\n\n### %s ### \nTraining Time: %s s' % (alg_name, round(t2-t1,5)))
y_pred = model.predict(train_feature)
accuracy =  round(accuracy_score(train_labels, y_pred) * 100,2)
print ("\tAccuracy on Train: %s" % (accuracy))
y_pred = model.predict(test_feature)
accuracy =  round(accuracy_score(test_labels, y_pred) * 100,2)
print ("\tAccuracy on Test: %s" % (accuracy))
```

Una volta allenato, il modello è pronto per essere testato. Viene importato un nuovo set di immagini ai cui punti non è stata assegnata alcuna label e questo è sottoposto alle stesse tecniche di elaborazione precedentemente utilizzate: trasformazione in array, scambio delle dimensioni, conversione da RGB a scala di grigi e normalizzazione su 16 livelli. È scelta una slice delle 275 importate ed è nuovamente eseguita la funzione *model.fit(train_feature, train_labels)*. Attraverso la riga di codice

rs = widgets.RectangleSelector(ax, onclick_prev, drawtype = 'box')

è richiamata la funzione *onclick_prev()*, così definita:

```
def onclick_prev(eclick, erelease):
    previsioni = []
    text_prev = ' '
    if eclick.xdata + size < erelease.xdata:
        for i in range(int(eclick.xdata)+size//2,int(erelease.xdata)-size//2,step):
            for j in range(int(eclick.ydata)+size//2,int(erelease.ydata)-size//2,step):
                x = int(eclick.xdata) - size//2
                y = int(eclick.ydata) - size//2
                g = slice_number - size//2
                w = x + size
                h = y + size
                l = g + size
                feat = calculate_features(image[y:h,x:w,g:l], levels)
                previsioni.append(model.predict(feat)[0])
        previsioni = np.asarray(previsioni)
        tot = previsioni.shape[0]
        bone_perc = np.count_nonzero(previsioni == 'bone') / tot * 100
        background_perc = np.count_nonzero(previsioni == 'background') / tot * 100
```

```
        organ_perc = np.count_nonzero(previsioni == 'organ') / tot * 100
        text_prev = [bone_perc, '% bone', background_perc, '% background', organ_perc, '% organ' ]
        ax.add_patch(Rectangle((eclick.xdata- size//2, eclick.ydata- size//2), erelease.xdata-eclick.xdata,
                                                    erelease.ydata-eclick.ydata, alpha = 0.1, facecolor = 'b'))
        ax.text((eclick.xdata+erelease.xdata)//2, (eclick.ydata+erelease.ydata)//2, text_prev, color="red",
                                                                        fontsize = 15)
        fig.canvas.draw()
    else:
        x = int(eclick.xdata) - size//2
        y = int(eclick.ydata) - size//2
        g = slice_number - size//2
        w = x + size
        h = y + size
        l = g + size
        feat = calculate_features(image[y:h,x:w,g:l], levels)
        ax.add_patch(Rectangle((x-size//2, y-size//2), size, size, alpha = 0.1, facecolor = 'b'))
        ax.text(x, y, model.predict(feat)[0],  color="red", fontsize = 15)
        fig.canvas.draw()
```

Tale funzione permette di cliccare con il mouse sulla slice dell'immagine visualizzata per ottenere la label a cui il punto appartiene. Così come per il labelling, è possibile selezionare un'area dell'immagine oltre a singoli punti. In questo caso, che si verifica se *eclick.xdata + size < erelease.xdata*, per ciasun pixel ogni tre (numero definito da *step*) all'interno della regione individuata dal trascinamento del mouse, sono inizializzate le variabili x, y, g, w, h, l con i vertici del cubo di lato *size* (impostato a 5) che ha come centro il pixel in questione, viene utilizzata tale regione per il calcolo delle features e viene aggiunto all'array *previsioni* (attraverso la funzione *append()*) la label di ciascuno di tali punti (*model.predict(feat)*). Una volta eseguiti i precedenti comandi per tutti i pixel voluti, sono calcolate le percentuali per ogni label. Tali percentuali sono poi visualizzate in output. Nel caso della selezione di un unico punto, invece, sono inizializzate le variabili x, y, g, w, h, l con i vertici del cubo di lato *size* (impostato a 5) che ha come centro il punto cliccato, viene visualizzata la label restituita da *model.predict(feat)*. Un esempio è riportato in Figura 26.

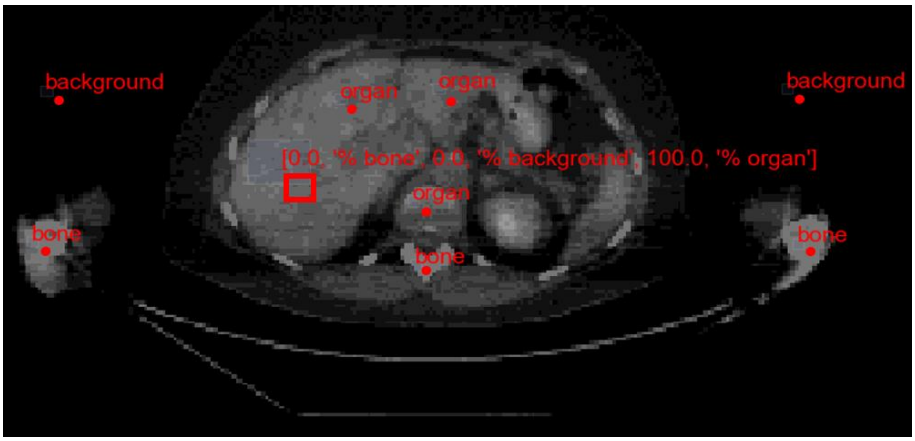

Figura 26. Esempio del testing del modello allenato. Nel caso della selezione di un unico punto, viene visualizzata la labe restituita dalla funzione model.predict(), nel caso della selezione di un'area dell'immagine, vengono visualizzate le percentuali delle label calcolate per i pixel contenuti nell'area.

Capitolo 5. Sviluppi futuri

Alcune migliorie possono essere apportate al codice per affinare e rendere più specifica la classificazione:

- Ampliamento del database utilizzando immagini diverse e un maggior numero di punti
- Creazione di nuove classi per rendere più specifica la classificazione

È inoltre possibile implementare il calcolo in parallelo delle features per diminuire ulteriormente i tempi computazionali e provare a calcolare la media delle features ottenute da ciascuna matrice di co-occorrenza nelle varie direzioni (invece che estrarre le 14 features dalla media delle matrici di co-occorrenza nelle varie direzioni) per verificare quale dei due metodi restituisca risultati migliori.

Conclusioni

L’oggetto di questa tesi è stata la creazione e l’implementazione di un codice Python per la classificazione dei tessuti all’interno di un’immagine medica in formato DCOM. A tale scopo, sono state adottate le immagini mediche PET-CT fusion per il loro vantaggio nel fornire informazioni sia anatomiche che funzionali. Sono state complessivamente importate e analizzate 6 immagini (set di slices): 5 di queste sono state utilizzate per il labelling e la rimanente parte è stata usata per la predizione del modello. Sono state presentate in dettaglio le modalità con cui le funzioni matematiche alla base dell’estrazione delle features di Haralick sono state utilizzate al fine di riconoscere i diversi tipi di tessuti. Sono stati illustrati gli step progettuali della creazione dei dataset delle label e delle features e della selezione e allenamento del miglior modello per la classificazione. L’approccio seguito ha permesso di classificare diversi pixel di un’immagine DCOM nelle classi “bone”, ossa, “organ”, organi, e “background”, sfondo. Possibili future implementazioni della metodologia adottata comprendono la creazione di nuove classi per affinare la classificazione dei diversi tessuti.

Bibliografia

[1] Wikipedia, "*Fluorodesossiglucosio*", tratto da https://it.wikipedia.org/wiki/Fluorodesossiglucosio.

[2] L. Kostakoglu, R. Hardoff, R. Mirtcheva, J. S. Goldsmith, "*PET-CT Fusion Imaging in Differentiating Physiologic from Pathologic FDG Uptake", RadioGraphics Vol. 35, No. 5,* 2004.

[3] R. M. Haralick, K. Shanmugam, I. Dinstein, "Textural Features fo Image Classification", *IEEE Transactions on systems, man, and cybernetics, VOL. SC-3, No. 6,* 1973.

[4] Wikipedia, "*Machine Learning*", tratto da https://en.wikipedia.org/wiki/Machine_learning.

[5] P. Gupta, "*Decision Trees in Machine Learning*", A Medium Corporation.

[6] S. Patel, "*Chapter 5: Random Forest Classifier*", A Medium Corporation.

[7] V. Morde, "*XGBoost Algorithm*", A Medium Corporation.

[8] XGBoost Documentation, tratto da https://xgboost.readthedocs.io/en/latest/index.html#.

[9] S. Patel, "*Chapter 2: SVM (Support Vector Machine) – Theory*", A Medium Corporation.

[10] S. Patel, "*Chapter 4: K Nearest Neighbors Classifier*", A Medium Corporation.

[11] Scikit Learn, "*Stochastic Gradient Descent*", tratto da https://scikit-learn.org/stable/modules/sgd.html#sgd.

[12] R. Gandhi, "*Naive Bayes Classifier*", A Medium Corporation.

[13] N. K. Kain, "*Understanding of Multilayer perceptron (MLP)*", A Medium Corporation.

[14] A. Pant, "*Introduction to Logistic Regression"*, A Medium Corporation.

[15] S. Sawla, "*Linear Discriminant Analysis*", A Medium Corporation.

[16] P. Gupta, "*Cross-Validation in Machine Learning*", A Medium Corporation.