ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

SVILUPPO DI UN SISTEMA DI RICONOSCIMENTO DI STENOSI CORONARICHE NON GRAVI

Elaborato in
Programmazione Di Applicazioni Data Intensive

Relatore
Prof. Gianluca Moro

Presentata da
Matteo Scala

Prima Sessione di Laurea
Anno Accademico 2020 – 2021

PAROLE CHIAVE

Coronarografia

Stenosi Coronariche

Deep Neural Networks

Machine Learning

Python

Introduzione

L'Intelligenza Artificiale è un campo dell'informatica che da tempo si afferma come valido strumento alternativo per la risoluzione di problemi tipicamente riservati esclusivamente all'intelletto umano.

Se in principio gli algoritmi sfruttati nel campo dell'Intelligenza Artificiale erano basati su insiemi di regole codificate da esperti del dominio di applicazione dell'algoritmo, con l'arrivo del secondo millennio questo approccio è stato superato in favore di algoritmi che sfruttano grandi quantità di dati ed elevata potenza di calcolo per fare scelte ottimali. Un esempio di questo approccio può essere Deep Blue, che nel 1996, anche grazie ad un database di 4mila aperture e un'architettura che permetteva 11 GFLOPS fu la prima macchina a vincere una partita a scacchi contro un grande maestro.

Col passare degli anni, l'aumentare degli investimenti e della ricerca, questo approccio ha portato alla strutturazione del campo dell'Apprendimento Automatico (Machine Learning, in inglese) dal quale sono scaturiti numerosi avanzamenti che hanno influenzato una moltitudine di ambiti: dall'agricoltura di precisione alla traduzione automatica, dal riconoscimento di frodi con carte di credito alla farmaceutica, dal marketing alla visione artificiale e molti altri, inclusa la medicina.

Questo lavoro si concentra su proprio questioni relative al campo della medicina. In particolare si occupa di provare a riconoscere se le stenosi coronariche di un paziente sono gravi o meno attraverso l'uso di angiografie coronariche invasive e tomografie coronariche angiografiche; in maniera da diminuire delle angiografie coronariche invasive effettuate su pazienti che non ne hanno davvero bisogno.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Panoramica del Problema

1.1 L'aspetto clinico

Per capire meglio il problema che viene affrontato, è utile soffermarsi (brevemente, siccome non è il mio campo) sugli aspetti clinici del problema e dare qualche informazione sul contesto ed il dominio nel quale si sviluppa la soluzione.
Il problema sta nel fatto che con l'avanzare dell'età aumenta il rischio che si formino stenosi (e che queste siano gravi) nelle arterie coronariche. Gli effetti delle stenosi possono variare fino a causare morte cardiaca improvvisa. Per poter valutare le stenosi si può fare uso di tomografia angiografica coronarica, che è un esame poco invasivo e relativamente leggero, oppure della angiografia coronarica invasiva che è maggiormente complessa ma considerata gold standard per la valutazione delle stenosi.

1.1.1 Struttura dei vasi coronarici

I vasi coronarici sono i vasi che si occupano di irrorare il cuore ed i muscoli che lo compongono; avvolgono il miocardio e portano il sangue che permette al cuore di compiere il ciclo cardiaco di sistole e diastole. Notare che non si sta parlando delle arterie e della vena polmonare che portano il sangue dentro ad atri e ventricoli.

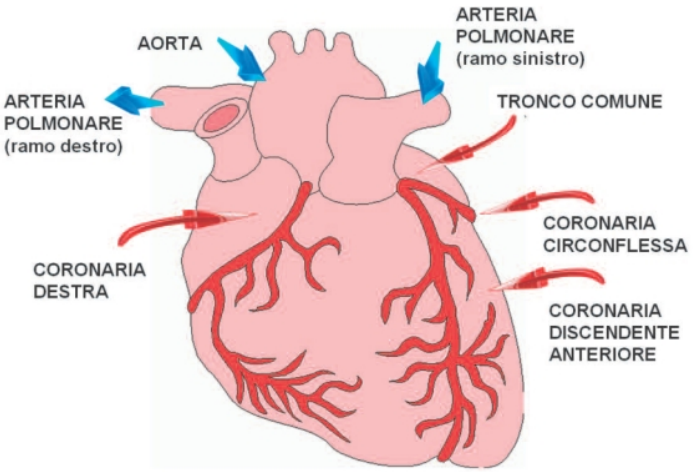

Figura 1.1: Rappresentazione artistica di un cuore con le coronarie evidenziate.
Fonte: blogspot.com

Le arterie coronariche si diramano a partire dall'aorta ascendente. I due vasi che partono dall'aorta costituiscono la *arteria coronaria destra* e *arteria coronaria sinistra*.

La prima, abbreviata CDX, può essere suddivisa nei tratti prossimale, medio e distale (a seconda della distanza dall'aorta) e poi (dopo il *crux cordis*) emette il tratto interventricolare posteriore.

La coronaria sinistra invece parte con il *tronco comune* che si dirama in tre vasi: l'arteria interventricolare anteriore (IVA), il ramo circonflesso (CFX) ed il ramo intermedio (che in certe persone non si sviluppa e lascia il tronco comune con due sole diramazioni). Sia l'IVA che il CFX si suddividono nei tratti prossimale, medio e distale (sempre a seconda della distanza dall'inizio della diramazione); hanno anche diramazioni dette settali, diagonali o marginali. Fanno parte delle arterie coronariche anche l'arteria mammaria interna destra (AMIS) e l'arteria mammaria interna sinistra (AMID).

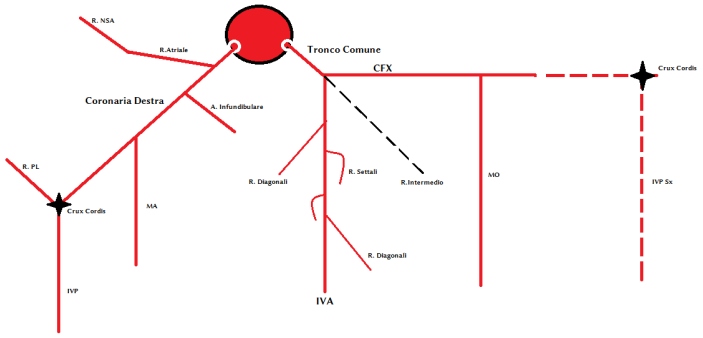

Figura 1.2: Rappresentazione schematizzata delle coronarie.
Fonte: dimensioneinfermiere.it

1.1.2 Stenosi coronariche

Le stenosi delle arterie coronariche sono lesioni che si formano, appunto, nelle arterie coronariche. Le probabilità delle stenosi di presentarsi aumentano soprattutto con l'età, la sedentarietà, il diabete (più altri fattori di rischio) e sono causate da accumuli di grassi e calcio che si depositano sulle pareti delle coronarie causando l'indurimento ed il restringimento delle pareti del vaso nella parte che segue l'ostruzione.

Le stenosi coronariche posso avere una serie molto diversa di effetti, si va dalla completa assenza di sintomi (soprattutto se la stenosi è lieve o localizzata alla fine dell'arteria coronaria) alla morte cardiaca improvvisa, passando per sintomi intermedi come dolori toracici, infarti, necrosi.

Per questo è importante riconoscere le stenosi e valutarle accuratamente per decidere se è il caso di trattarle, quando trattarle e come (in maniera farmacologica o intervenendo) trattarle.

1.1.3 Esami di controllo

Per valutare le stenosi delle arterie coronarie si usano principalmente due procedure.

La *angiografia coronarica* (o coronarografia) che è una procedura invasiva che consiste nell'inserimento di un catetere all'interno del paziente tramite due vie d'accesso principali ovvero l'arteria radiale (nel braccio) o l'interno dell'arteria femorale (nella gamba). Il catetere viene poi spinto all'interno delle arterie fino al cuore ed una volta messo in posizione, attraverso viene emesso il

Figura 1.3: Frame di una radiografia dinamica di una coronarografia.
Fonte: cardioumg.it

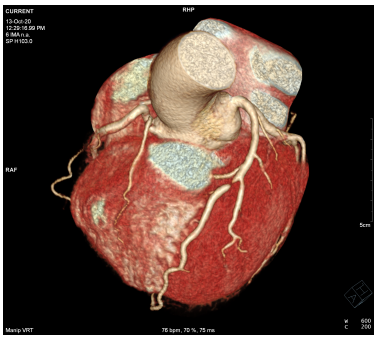

Figura 1.4: Ricostruzione tridimensionale di cuore dopo una TAC.
Fonte: grupposandonato.it

mezzo di contrasto che permette di fare risaltare le arterie ai raggi X. Questa procedura ha il vantaggio di permettere anche la valutazione di disfunzioni nei movimenti dei muscoli o nella funzionalità delle valvole del cuore. Inoltre, durante l'operazione, può anche essere effettuata un'angioplastica nel caso si rilevino stenosi particolarmente critiche.

In alternativa, può essere effettuata una *tomografia angiografica coronarica*. In questo caso l'operazione non è invasiva, viene inserito nel paziente il mezzo di contrasto per iniezione da un vaso periferico del corpo o per assunzione orale, in poco tempo il mezzo arriva alle arterie coronariche e si può usare l'apparecchiatura per la TAC per analizzare le arterie del paziente. La procedura è molto breve e comporta solo il rischio per le radiazioni ionizzanti assorbite dal paziente, che grazie agli ultimi sviluppi tecnologici, sono sempre meno [1].

Tra le due procedure, quella riconosciuta come più affidabile è la coronarografia [2], ma è anche quella che comporta i rischi più alti, in quanto richiede l'inserimento all'interno dei vasi del paziente di un catetere, con tutte le complicanze che ne conseguono.

La TAC invece è considerata maggiormente utile per l'esclusione di stenosi [3] [4]. Se si riuscisse a "migliorare" la valutazione di stenosi non gravi anche con la TAC, si avrebbero numerosi vantaggi sia per i pazienti che per gli ospedali.

1.2 Applicazioni correnti di metodi predittivi

Negli ultimi anni, è in aumento l'applicazione di tecniche di apprendimento automatico al campo della medicina cardiovascolare [5]. Ciò, a causa alla sempre

maggiore disponibilità di immagini e dell'abilità degli algoritmi impiegati di gestire grandi quantità di dati.

In particolare, gli studi più recenti hanno portato ad importanti risultati nella segmentazione automatica delle arterie [6], riconoscimento di stenosi, occlusioni, calcificazioni, trombosi e dissezioni [7] ed anche al riconoscimento della gravità delle stenosi [8] (con coronarografie), fino alla previsione del decorso finale delle stenosi [9] [10].

Nonostante i progressi ed i vantaggi, bisogna ricordare che applicare algoritmi di apprendimento automatico ha comunque alcuni lati negativi come la forte dipendenza dalla qualità dei dati a disposizione e spesso (a seconda degli algoritmi scelti) risulta complicata l'interpretazione dei risultati finali [5].

Capitolo 2

Il Progetto

2.1 Obiettivo del progetto

In questo progetto l'obiettivo è quello di correlare la gravità di stenosi rilevate con la TAC con la gravità delle stenosi rilevate con coronarografia, migliorando la capacità della TAC di rilevare stenosi non gravi.

La correlazione sarà cercata approcciando il problema come una **classificazione**, associando ad ogni lettura (sia da TAC che da coronarografia) una classe "stenosi grave" o "stenosi non grave".

Tramite l'utilizzo di tecniche di machine learning, si cercherà soprattutto di riconoscere le letture di stenosi che appaiono come "gravi" se effettuate con la TAC ma che risultano "non gravi" se effettuate con la coronarografia.

2.2 I dati

I dati utilizzati in questo lavoro sono stati raccolti dall'ospedale Villa Maria Cecilia e dalla azienda Halnet Srl. Il primo ha messo a disposizione operatori esperti (emodinamisti, radiologi e cardiologi) che hanno valutato le stenosi con metodo qualitativo (ovvero con una valutazione visiva, senza l'utilizzo di software [11]), la seconda ha gestito la raccolta e l'organizzazione delle immagini e delle valutazioni.

I dati raccolti riguardano complessivamente 218 pazienti anonimizzati; per ognuno sono state registrate le gravità delle stenosi per 21 tratti delle arterie coronarie, secondo le linee guida del SCCT [12], ovvero con una valutazione "intera" da 1 a 6 inclusi, dove ad ogni numero è associato un determinato grado di ostruzione (ad esempio: 1 corrisponde ad un ostruzione dello 0%, 4

Paziente	TC	IVA_PROX	IVA_MEDIA	IVA_DISTALE
Anon1	1	3	5	1

Tabella 2.1: Esempio (tagliato) di valutazione: con locazione della stenosi e relativa entità

corrisponde ad un ostruzione tra il 50% ed il 69%). Per ogni paziente sono state eseguite 2 letture della TAC e due letture della coronarografia, per un totale di 4 letture fatte tutte da esperti differenti.

Nota: i dati sono stati processati ed elaborati tramite la libreria Pandas [13] di Python, che offre prestazioni e funzioni molto ottimizzate (grazie all'implementazione in C di una parte sostanziale della libreria) rispetto a strutture dati implementate solamente con Python.

2.2.1 Preprocessamento dei dati

Siccome è necessario poter associare le letture avvenute tramite TAC e le letture avvenute con la coronarografia paziente per paziente e siccome per alcuni pazienti non sono state valutate le stenosi; il primo passo è stato quello di eliminare tutte le letture effettuate con una metodologia per le quali manca la corrispettiva lettura con la metodologia alternativa. Se, ad esempio, un paziente non ha le valutazioni relative alla TAC perché l'esame ha rilevato solo il *calcium-score*, allora anche le letture dello stesso paziente effettuate con la coronarografia sono state eliminate; lasciando 165 letture in totale.

È stata effettuata una rapida analisi intra-metodica per controllare la discrepanza presente tra le letture dei due operatori. Le letture della TAC erano identiche nel 40% dei casi, se si includono differenze di una sola unità la similarità aumenta fino al 60%; spesso differenze più ampie erano dovute al diverso posizionamento delle stenosi tra i due operatori (non nella valutazione dell'intensità). Un ragionamento analogo è stato fatto per la coronarografia; in questo caso, anche a causa della diversa metodica, la quantità di letture identiche risulta molto maggiore: 86%. Alla luce di queste similarità si è proceduto considerando solamente letture ottenute da un solo operatore.

In fine si è proseguito associando ad ogni istanza una classe. Una lettura è considerata grave se è presente almeno una stenosi grave. Una stenosi è grave se l'occlusione è maggiore del 70% (che corrisponde ad un valore di `5` nel

dataset) o se la stenosi si trovava nel tronco comune e causava un occlusione del 50% (ovvero un valore di `4`). Ad ogni istanza è stata assegnata la classe "non grave" se la lettura della TAC appariva lieve o se la lettura della coronarografia appariva lieve, così da ottenere che le letture che risultavano gravi con una TAC ma lievi con la coronarografia venissero comunque classificate come "con gravi". Questa suddivisione ha portato ad una dataset quasi perfettamente *bilanciato* tra istanze positive e negative, con 83 valutazioni classificate come critiche e 82 valutazioni classificate come non critiche [14].

Si è anche provato a sfruttare lo *sdoppiamento*. Lo sdoppiamento consiste nel separare tutti i pazienti con letture differenti da quelli con letture identiche; assicurarsi che diverse letture di stessi pazienti venissero mantenute nello stesso set di dati e utilizzate entrambe o per l'addestramento o per la valutazione dei modelli predittivi; effettuando in fine tutte le valutazioni relative a pazienti sdoppiati tramite un sistema che riunisse le letture sdoppiate e che assegnasse una classe tramite *voting*.

2.3 I metodi predittivi

Come già indicato all'inizio del capitolo, il problema è stato modellato come un problema di classificazione; dove ogni istanza corrisponde alla lettura di un esame TAC e ogni classe è "non grave" (positiva) o grave (negativa). Le opzioni che nel tempo sono state sviluppate nel campo dell'apprendimento automatico per risolvere questa tipologia di problemi sono tante:

- **Regressione Logistica** pensata già nel 1944 [15] che permette di associare alle variabili indipendenti dell'osservazione (valutazione di stenosi in questo caso) dei coefficienti che combinati restituiscono la probabilità dell'osservazione di appartenere ad una certa classe.
- **Support-vector Machine** [16] che dividono le istanze cercando l'iperpiano che meglio separa le istanze più difficili da classificare (ovvero quelle che si troveranno più vicine all'iperpiano) dette appunto *support vector*.
- **K-nearest neighbors** [17] che è un metodo più semplice che consiste nel classificare l'istanza sulla base della classe più presente tra le `K` istanze (già osservate) più vicine.
- **Alberi di decisione** [18] (in particolare CART [19]) dove l'obiettivo è quello di apprendere semplici regole sulle quali basarsi per la classifica-

zione delle istanze. Questi alberi hanno il vantaggio di essere facilmente interpretabili, ma lo svantaggio di avere una forte tendenza all'overfitting (adattarsi bene ai dati dell'addestramento, ma male ai dati di test); per questa ragione più spesso si utilizzano **Random Forest**, ovvero insiemi di alberi decisionali, creati a partire da dataset leggermente diversi, che apprendendo criteri differenti offrono una maggiore capacità di generalizzare e migliori accuratezze [20] [21].

- **Gradient Boosted Tree** [22] che applica un approccio simile a quello delle Random Forest, ma che differisce per la maniera in cui sono costruiti gli alberi e per la maniera con cui le decisioni vengono combinate. Esistono diverse implementazioni di questo approccio: **Extreme Gradient Boosting** (XGBoost) [23] è una libreria che migliora le performance portando una automatica selezione delle features, parallelizzazione dell'addestramento ed altro; **CatBoost** [24] che supporta nativamente features categoriche; oppure **Light Gradient Boosting Machine** [25] che ottimizza ulteriormente la costruzione degli alberi utilizzando istogrammi ed altre tecniche.

- **Multilayer Perceptron** (MLP) [26] anche conosciuto come rete neurale artificiale; che combina più **Percettori** [27] organizzati in più strati dove l'ultimo strato corrisponde all'output del MLP. In questa tipologia di algoritmi ogni percettore apprende una separazione delle istanze secondo un iperpiano, la combinazione di questi iperpiani permette molta flessibilità ed in questo lavoro ha portato ai migliori risultati.

- **TabNet** [28] è una particolare rete neurale artificiale costruita con un'architettura che si presta all'applicazione su dati di tipo tabellare. Al suo interno vengono fatte consecutive selezioni di features tramite il meccanismo dell'*attenzione* [29] poi processate e ricombinate per dare la classificazione finale.

Tutti questi metodi costituiscono un sottoinsieme dei metodi di *apprendimento supervisionato* e sono stati citati perché sono stati utilizzati in questo lavoro.

Tutti i metodi di apprendimento automatico hanno una cosa in comune: dipendono da una funzione, detta *loss*, che misura quanto la predizione fatta si differenzia dalla risultato esatto osservato. L'apprendimento avviene quando gli

algoritmi riescono a produrre previsioni che abbassano il valore della funzione d'errore.

Il caso del k-nearest neighbors è particolare perché durante l'addestramento non vengono appresi dei parametri, ma piuttosto memorizzate le istanze; rimane il fatto che la classe predetta y, per una determinata istanza x, è quella che minimizza il numero di altre istanze appartenenti alla classe opposta, prese dall'insieme delle k più vicine.

$$y = argmin_C \sum_{i:x_i \in N_k(X,x)} \delta(y_i, \mathcal{C}) \tag{2.1}$$

Nel caso della regressione logistica, la funzione *loss* minimizza il risultato della funzione $softmax$; la quale dipende dalle coppie x y utilizzate per l'addestramento e dai parametri b e w che varieranno (e saranno quindi appresi) durante l'addestramento.

$$\min_{\mathbf{b},\mathbf{w}} \sum_i log(1 + e^{-y_i * h_w(x_i)}) \tag{2.2}$$

Invece le support vector machines minimizzano la norma euclidea dei coefficienti w che definiscono l'iperpiano. Diversa ancora è la funzione minimizzata dagli alberi decisionali. In questo caso si misura l'errore (e quindi si minimizza) come la media pesata dell'indice di eterogeneità (impurità) di Gini [30] che è, per ogni ramificazione, 1− la somma dei quadrati delle frequenze con cui una classe appare nelle foglie della ramificazione.

$$impurity = 1 - \sum_i^C p_i^2 \tag{2.3}$$

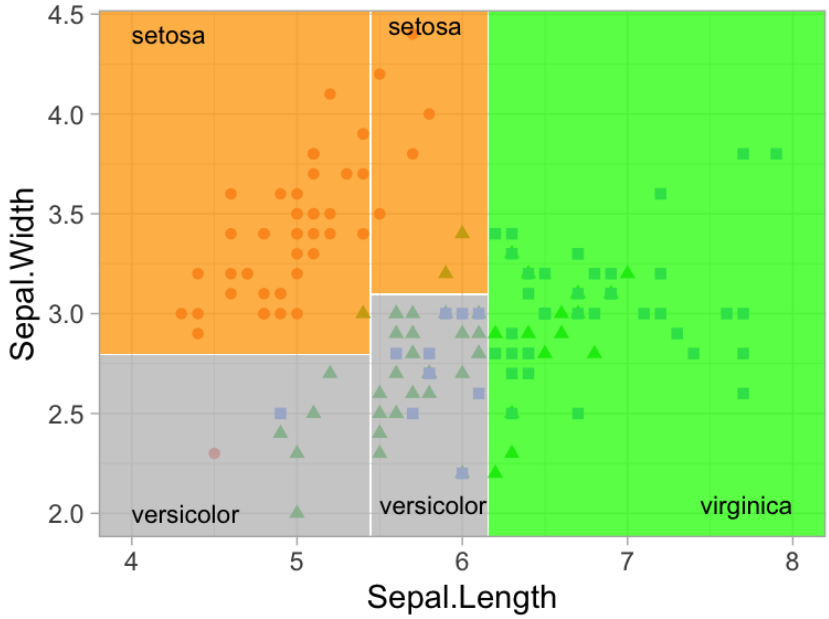

Figura 2.1: Classificazione di istanze bidimensionali con alberi.
Fonte: github.io

Anche nel caso dei *gradient boosted trees* si minimizza l'opposto di una funzione di verosimiglianza che misura la bontà di una predizione p iniziale rispetto ai dati osservati y

$$\sum_{i=1}^{N}(y_i * log(p_i)) + ((1 - y_i) * log(1 - p_i)) \tag{2.4}$$

L'addestramento di questi modelli, avviene con metodi iterativi, che permettono miglioramenti graduali del valore della funzione di *loss* tramite l'aggiornamento dei parametri che caratterizzano il modello.

Uno di questi metodi è detto discesa stocastica del gradiente [31]. Si usa nelle support vector machines, nella regressione logistica e nelle reti neurali artificiali (quindi anche nei multilayer perceptron). Questo metodo non è impiegato nel k-nearest neighbors che non ha parametri da apprendere e non è impiegato nella costruzione di alberi che vengono costruiti combinando diverse possibili decisioni sulla base dell'impurità di Gini.

Per spiegare meglio come funziona la discesa del gradiente è utile fare un esempio. L'esempio più chiaro può essere portato prendendo come riferimento la funzione della regressione logistica univariata lineare. Il *gradiente* di una

funzione (in questo caso della funzione *loss*) è un vettore le cui componenti sono le derivate parziali della funzione stessa calcolate rispetto ai parametri di riferimento. Nel caso della regressione logistica univariata lineare la funzione di *loss* è:

$$\min_{\mathbf{b},\mathbf{w}} \sum_i log(1 + e^{-y_i*(b+w*x_i)}) \tag{2.5}$$

e di conseguenza, il gradiente è

$$\sum_i^N -\frac{y_i}{e^{y_i*(b+w*x_i)}+1}; \sum_i^N -\frac{x_i * y_i}{e^{y_i*(b+w*x_i)}+1} \tag{2.6}$$

rispettivamente per il parametro b e per il parametro w.

Si possono quindi utilizzare le componenti del gradiente, opportunamente scalate di un fattore detto *learning rate*, per aggiornare i parametri e quindi per diminuire l'errore commesso dal modello nel classificare i dati.

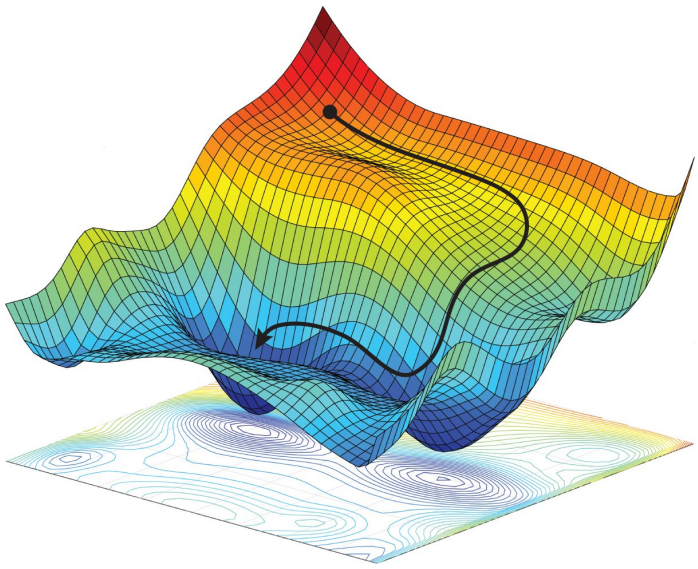

Figura 2.2: Illustrazione della discesa del gradiente di una funzione in due parametri, dove l'altezza è il valore della funzione *loss*.
Fonte: caltech.edu

Questo metodo di minimizzazione dell'errore si applica però ad una sola funzione d'errore. Quando si deve applicare ad un multilayer perceptron, che è composto da più percettori e ad ognuno bisogna associare la sua "responsabilità" nel generare l'errore totale della rete, l'algoritmo diventa più complicato. In questi casi si combina la discesa del gradiente con l'algoritmo della retropropagazione dell'errore (*backpropagation*). [32].

In certe situazioni, soprattutto per modelli semplici come la regressione logistica e le support vector machines, non è possibile minimizzare l'errore durante l'addestramento. Questo è spesso causato dalla distribuzione delle istanze. Modelli predittivi come la regressione logistica e le support vector machines sono fatti per cercare un iperpiano che separi linearmente le istanze; se le istanze sono distribuite in maniera da non essere separabili linearmente, la discesa del gradiente troverà comunque un minimo della funzione d'errore, ma da questo minimo non si otterranno buoni risultati.

Questo problema può essere risolto applicando alle istanze del dataset delle trasformazioni che le mappino in spazi a più alta dimensionalità, dove diventa possibile la separazione lineare cercata dai modelli sopracitati. Questa soluzione viene però ad un costo: le trasformazioni da applicare sono spesso computazionalmente molto impegnative. Per questa ragione si ricorre al "kernel trick" [33]. Esso permette di evitare l'esplicita trasformazione delle istanze del dataset, sostituendola con il calcolo di una matrice (con tante righe e colonne quanto il numero di elementi del dataset) che riportata la relazione tra ogni coppia di istanze nello spazio in cui queste sarebbero portate dalla trasformazione originale. Questo "trucco" permette anche di rappresentare relazioni che sarebbero impossibili se si ricorresse alla trasformazione esplicita, per esempio trasformazioni che porterebbero le istanze in uno spazio ad infinite dimensioni, si veda il caso del kernel della funzione radiale di base [34].

$$K(x,y) = e^{-\frac{d(x,y)^2}{2l^2}} \rightarrow e^{-\frac{1}{2}(x-y)^2} = e^{-\frac{1}{2}(x^2+y^2)}e^{xy} \tag{2.7}$$

$$e^{xy} = 1 + xy + \frac{1}{2}xy^2 + \frac{1}{3!}xy^2 + ... + \frac{1}{\infty}xy^\infty \tag{2.8}$$

$$e^{xy} = (1, x, \sqrt{\frac{1}{2}}x^2, ..., \sqrt{\frac{1}{\infty}}x^\infty) * (1, y, \sqrt{\frac{1}{2}}y^2, ..., \sqrt{\frac{1}{\infty}}y^\infty) \tag{2.9}$$

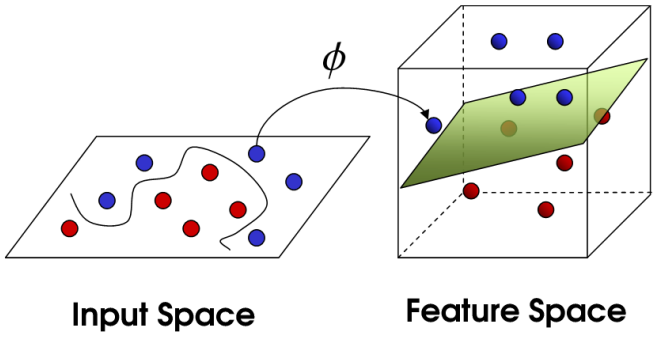

Figura 2.3: Esempio di una trasformazione che aggiunge una dimensione alle istanze.
Fonte: medium.com

Altri problemi che si possono verificare con questi modelli sono *underfitting* e *overfitting*. Si verifica *undefitting* quando il modello che si è scelto non è in grado di adattarsi ai dati, può essere causato dai problemi prima citati che riguardano la distribuzione dei dati o può essere causato semplicemente da una cattiva scelta del algoritmo, che non è in grado di rappresentare la complessità dei dati. Si verifica *overfitting* quando il modello perde la capacità di generalizzare e si adatta troppo bene ai dati che utilizza durante l'addestramento ma male ai dati mai osservati. Quando si verifica questo iperadattamento è bene cercare di smorzare la capacità del modello di adattarsi ai dati applicando della *regolarizzazione*.

La regolarizzazione può essere applicata a qualsiasi modello e va ad inserirsi all'interno della misura dell'errore che si sta minimizzando; in altre parole, si accetta una piccola quantità di errore sulla parte delle osservazioni usata per l'addestramento, per ottenere una riduzione dell'errore su i dati mai visti; anche se bisogna tenere presente che con la regolarizzazione si può esagerare e perdere accuratezza anche su i dati mai osservati.

Ad esempio, il numero k di vicini da considerare in un modello k-nearest neighbors può essere aumentato per cercare più regolarizzazione, ma esagerando si ottiene semplicemente una predizione che equivale sempre alla classe più presente nei dati di addestramento; oppure la profondità massima di un albero decisionale può essere limitata per ridurre l'adattamento. Invece, nelle reti neurali artificiali, l'*overfitting* può essere causato da una eccessiva dipendenza della rete da un singolo nodo e per regolarizzarla si inseriscono particolari "strati", detti *dropout*, che annullano (azzerano) l'output di un nodo durante l'addestramento (non durante l'inferenza) in maniera casuale costringendo la

rete a dipendere da tutti i nodi.

Quando si tratta di regressione logistica, la regolarizzazione avviene penalizzando (quindi aumentando il valore della funzione *loss*) i modelli che presentano parametri w con valori elevati. Esistono due tipi di regolarizzazione; essi aggiungono alla funzione d'errore (e poi alle derivate nel gradiente) la *norma 1* o la *norma 2* del vettore dei parametri, forzando la discesa del gradiente in un minimo che ha rispettivamente o molti parametri annullati (uguali a 0) o una piccola piccola differenza tra i parametri. Queste due tipologie di regolarizzazione sono spesso usate in combinazione una con l'altra, bilanciate da un parametro α che sposta la rilevanza di una norma o dell'altra nella funzione d'errore. Anche la regolarizzazione è opportunamente scalata per un fattore da impostare ad inizio addestramento.

2.3.1 I modelli di previsione

In questo lavoro, sono stati testati tutti i modelli elencati all'inizio del capitolo, ma solo alcuni si sono rivelati efficaci. Quelli che sono rivelati più efficaci sono stati il K-nearest neighbors, XGBoost, le Random Forest ed il multilayer perceptron; ma una combinazione di tutti è quella che è risultata la migliore in assoluto.

Data l'importanza che hanno avuto questo modelli, potrebbe essere opportuno un approfondimento sul loro funzionamento, su come sono stati addestrati e combinati.

Si può partire dal più semplice, ovvero k-nearest neighbors. Come già spiegato sopra, questo algoritmo classifica una istanza sulla base della classe delle k istanze che gli sono più vicine. Esistono pochi algoritmi per il recupero delle istanze più vicine ad una data, perciò è spesso impiegato un algoritmo *brute force* che calcola la distanza per tutte le N istanze in tempo $O(N^2)$. Questo approccio diventa velocemente intrattabile per grandi quantità di istanze. In alternativa si possono usare altri algoritmi che costruiscono strutture dati che permettono il recupero di istanze più vicine più velocemente. Questi algoritmi sfruttano l'idea che se c'è bisogno di calcolare la distanza di un punto A da altri due punti B e C, e se si sia già che B e C sono molto vicini, dopo aver calcolato la distanza tra A e B se si scopre che sono lontani, allora si può evitare il calcolo della distanza tra A e C assumendo che anche in questo caso la distanza sia ampia.

Uno di questi algoritmi sfrutta un *ball tree* [35] (una struttura dati ad albero pensata apposta per organizzare e partizionare spazi o elementi in uno spazio). Si usano le istanze dell'insieme dei dati di addestramento per costruire un albero binario che divide le istanze separandole con nodi che definiscono delle ipersfere (un'ipersfera per nodo) tali che tutte le istanze rimangono contenute all'interno di un'ipersfera. Quando vengono richieste le istanze più vicine ad una data, si scende l'albero fino al nodo che rappresenta l'ipersfera con centro più vicino all'istanza data e che contiene un numero maggiore di k di istanze. Tra le istanze all'interno dell'ipersfera si procede con una calcolo *brute force* delle distanze. Appoggiarsi ai *ball tree* permette di effettuare il calcolo in $O(log(N))$.

Un'altra variabile importante del k-nearest neighbors è data dalla distanza che si decide di calcolare tra ogni istanza. La distanza più comune (che è anche quella che è stata usata in questo lavoro) è quella detta "di Minkowski".

$$D_M(x, y, p) = \sqrt[p]{\sum_{i}^{N} |x_i - y_i|^p} \tag{2.10}$$

Questa distanza coincide con quella euclidea se p è uguale a 2. Anche se altre ricerche hanno mostrato l'efficacia di distanze più complesse, tipo quella "di Hassanat" [36], esse non sono state implementate perché il modello raggiungeva comunque buoni risultati.

Un altra tipologia di modelli che è risultata efficacie sono state le Random Forest. In questo caso, più che di un modello vero e proprio, si tratta più di una combinazione di modelli e rientra sotto il cappello delle metodologie *ensamble*.

Queste metodologie prendono decisioni sull'output finale combinando gli output di più modelli e si possono dividere in 3 categorie: il *bagging* costruisce diversi classificatori deboli differenziandoli per il dataset di addestramento dal quale partono, *boosting* costruisce in sequenza diversi classificatori sulla base degli errori commessi da quelli precedenti, *stacking* costruisce classificatori di diversi tipologie ed addestra un modello superiore che impara a classificare dalle predizioni dei modelli alla base.

Le random forest appartengono alla prima categoria: il dataset di train di partenza viene suddiviso in maniera random in tanti sottoinsiemi (non per forza disgiunti) quanti gli alberi decisionali che si vogliono costruire. Per costruire un albero decisionale si parte con la costruzione di tanti *stump* (alberi binari con un solo livello di profondità) quanti sono gli attributi booleani del dataset, se un attributo è continuo si ordina il dataset secondo i valori di quell'attributo

e si costruisce uno *stump* per ogni media di valori adiacenti; se un attributo è categorico tipicamente si trasforma con una codifica *one-hot* e si ricade nel caso degli attributi dicotomici. Per ogni albero iniziale si calcola l'impurità di Gini e si seleziona come radice dell'albero finale quello con il valore dell'impurità minore. Questo procedimento prosegue finché non si raggiunge la profondità massima o il numero massimo di foglie o non si separano perfettamente i dati. Certe volte, oltre a suddividere il dataset, si rimuovo in maniera random diverse *features* in diversi dataset per aumentare ulteriormente la capacità di generalizzare. Il valore predetto in fine dalla foresta coincide con la classe più predetta dagli alberi costruiti in precedenza [37]; in realtà l'implementazione di sci-kit learn [38] differisce leggermente in quanto effettua una media delle probabilità di appartenere ad una certa classe che vengono calcolate da ogni albero.

In questo lavoro è stato utile anche l'applicazione di XGBoost che implementa il metodo del *gradient boosting* in maniera estremamente efficiente. L'idea basilare di questo algoritmo è di approssimare il compito (funzione) della classificazione una con una serie di funzioni man mano più accurate.

Il primo passo è quello di fare una predizione iniziale, si calcola il logaritmo del rapporto tra il numero di casi positivi ed il numero di casi negativi. Questo valore si può interpretare attraverso la funzione sigmoidea come la probabilità di ogni istanza di appartenere alla classe che si sta predicendo e può essere usato per migliorare la predizione calcolando le differenze (errori di predizione) tra i valori osservati nel dataset (appartenenza o meno alla classe) e la predizione fatta all'inizio. Dopo aver calcolato gli errori si addestra un albero di regressione [19] (non di classificazione) che predica le differenze appena calcolate. Per combinare i valori di output dell'ultimo albero con la predizione fatta in precedenza è necessario trasformare i valori predetti dall'albero (perché l'albero predice delle differenze di probabilità, mentre la predizione iniziale è espressa come $log(odds)$) con la formula:

$$\frac{\sum_i^L p_i}{\sum_i^L p'_i * (1 - p'_i)} \tag{2.11}$$

Nella formula precedente L è il numero di elementi della foglia che si sta trasformando, p_i è il valore predetto dall'albero, p'_i è la probabilità precedentemente predetta. Dopo aver trasformato il valore di ogni foglia, averlo scalato per un opportuno *learning rate* e sommato alla prima previsione si possono calcolare le

nuove differenze rispetto ai valori osservati nel dataset per costruire un ulteriore albero e affinare ancora la previsione [22].

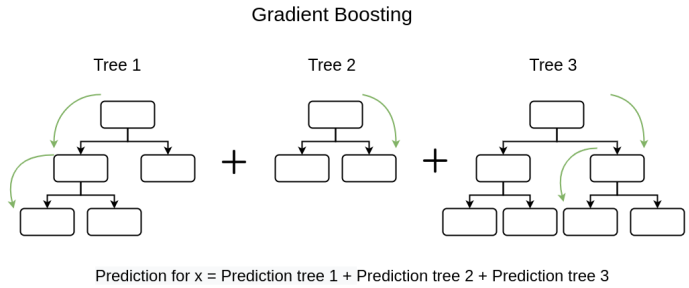

Figura 2.4: La predizione finale è data dalla somma di predizione che minimizzano gli errori.

Fonte: medium.com

L'ultima tipologia di modelli di previsione che si è rivelata molto utile e versatile è stata quella delle reti neurali artificiali, più in particolare dei multilayer perceptron. Il multilayer perceptron può essere immaginato come un grafo orientato organizzato in livelli consecutivi, dove gli archi connettono nodi solo se appartenenti a livelli adiacenti (questo non è sempre vero, in architetture più complesse alcuni archi possono saltare dei livelli [28], alcuni possono tornare al nodo di partenza [39] ed esistono molte altre possibili variazioni). I livelli che compongono i multilayer perceptron possono essere divisi in 3 categorie, a seconda ordine con il quale ricevono i dati all'interno dell'architettura possono essere: strati di *input* che ricevono i dati del dataset e li passano al resto della rete senza effettuare manipolazioni, *hidden* che tipicamente sono la maggioranza e sono gli strati che effettuano le elaborazioni dei dati e che portano l'informazione fino all'ultimo strato di *output* dove i dati sono preparati per essere presentati al esterno come risultato della classificazione. Ogni perceptron della rete deve contribuire alla corretta classificazione del dato fornito in input e per fare ciò adatta i propri parametri b e w proprio come una regressione logistica. L'apprendimento di questi parametri avviene calcolando il gradiente rispetto ai parametri considerando l'intera rete neurale come una composizione di funzioni.

Un parametro importante che influisce molto sulle prestazione del multilayer perceptron (e che non viene appreso durante l'addestramento) è la funzione che si sceglie di applicare in ogni perceptron ad ogni livello. Due esempi di funzioni molto impiegate sono la *rectified linear unit* che si calcola come $max(0, x)$ e *sigmoide* che si calcola come $\frac{1}{1+e^{-x}}$. Alcuni vantaggi della prima

sono che permette di risolvere il problema della scomparsa del gradiente [40] (che è un problema causato dalla concatenazione di gradienti che risultano tra 0 e 1 che vengono moltiplicati durante la backpropagation ed impediscono l'aggiornamento dei parametri) ed è più efficiente da calcolare, mentre un vantaggio della seconda può essere che avendo il codominio nell'intervallo $[0, 1]$ se posta negli strati di *output* della rete può agevolare l'apprendimento nel caso della classificazione. Altri parametri che influenzano le performance del multilayer perceptron sono il numero di strati, il numero di nodi in ogni strato, il numero di epoche di durata dell'addestramento, la quantità di istanze sulle quali calcolare l'errore prima di aggiornare i parametri interni della rete (*batch size*), il numero di addestramenti e tanti altri.

Il modello finale è stato ricavato alla fine di una serie di esperimenti che hanno riguardato una vasta gamma di modelli di previsione disponibili o compatibili con la libreria sci-kit learn. I modelli sono stati "ampliati" tramite l'inserimento all'interno di una *Pipeline* (una classe della libreria che permette di posporre e/o anteporre ai modelli di previsione delle manipolazioni sui dati) e sono stati affiancati a liste di possibili valori da associare agli iperparametri del modello e degli altri componenti della Pipeline. Tramite la classe *GridSearchCV* è stata effettuata una ricerca con convalida incrociata (*cross-validation* in inglese) delle combinazioni migliori di iperparametri di ogni Pipeline.

I risultati di questa esplorazione iniziale saranno esposti dopo la descrizione delle metriche attraverso le quali i modelli sono stati valutati.

2.3.2 Il modello di riferimento

Tutti i modelli addestrati sono stati confrontati e paragonati con un modello creato ad-hoc. Questo modello è stato creato per essere compatibile con la libreria sci-kit learn in maniera da permettere confronti più veloci.

Il modello preparato come riferimento simulava e semplificava una decisione che normalmente spetta ad un medico con un quadro clinico del paziente molto più ampio rispetto a quello offerto dal dataset. In questo caso, il modello classifica la situazione del paziente come "grave" o "non-grave" se le stenosi presenti ostruivano più del 70% del vaso in generale o se ostruivano più del 50% del tronco comune [14].

Tipicamente la valutazione della gravità di una stenosi prende in considerazione altri fattori come i sintomi manifestati dal paziente, lo stato di affatica-

mento del cuore, malattie già presenti (come il diabete), infarti precedenti e altro [14].

2.4 Definizione delle metriche rilevanti

Per confrontare i modelli addestrati tra di loro e rispetto al modello di riferimento è fondamentale definire delle metriche che possano rappresentare la bontà delle previsioni effettuate e che possano costituire dei metri di paragone adeguati.

Le metriche principali che si considerano quando il problema che si sta risolvendo riguarda la classificazione si ricavano dalla **matrice di confusione**.

La matrice di confusione è una matrice che ha un numero di righe e colonne pari al numero di classi che il modello deve imparare a prevedere. Per ogni colonna, la somma delle righe corrisponde al numero di istanze che il modello ha *previsto* come appartenenti alla classe relativa alla colonna presa in considerazione. Per ogni riga, la somma delle colonne corrisponde al numero di istanze *realmente* appartenenti alla classe relativa alla riga presa in considerazione. Se il problema di classificazione consiste di due sole classi, tipicamente si denota una delle due classi come "positiva" e l'altra come "negativa", ogni cella rappresenta una particolare tipologia di istanze: le entrate che compongono la diagonale contengono il numero di istanze classificate correttamente ("veri positivi" o TP e "veri negativi" o TN); la restante cella nella colonna della classe positiva rappresenta il numero di istanze classificate come positive che in realtà sono negative ("falsi positivi" o FP) e analogamente l'ultima cella rappresenta la quantità di istanze classificate come negative che sono in realtà positive ("falsi negativi" o FN).

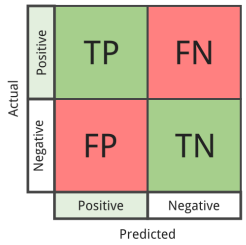

Figura 2.5: Illustrazione di una matrice di confusione con due classi.
Fonte: medium.com

Da queste 4 tipologie di istanze (veri positivi, falsi positivi, veri negativi, falsi negativi) si possono ricavare informazioni utili sul modello che si sta analizzando:

- **Accuratezza** è il rapporto tra il numero di istanze calcolate correttamente ed il numero totale di istanze: $\frac{TP+TN}{TP+TN+FP+FN}$
- **Precisione** che si può calcolare relativamente da una singola classe come $\frac{TP}{TP+FP}$ (nel caso della classe positiva) e rappresenta la capacità del modello di identificare correttamente le istanze appartenenti ad una certa classe. Si può fornire un calcolo aggregato delle precisione su entrambe le classi semplicemente calcolando una media delle due precisioni iniziali. Nel caso di questo lavoro, la precisione sulla classe positiva rappresenta la percentuale di persone classificate come "non gravi" che sono realmente "non gravi"; è ideale che sia più vicina possibile a 100%, perché avere falsi positivi significa sottovalutare l'entità delle stenosi di un paziente e non fornirgli l'eventuale trattamento di cui avrebbe bisogno.
- **Recupero** (o *recall* in inglese) è un altra metrica che può essere calcolata relativamente ad una singola classe come $\frac{TP}{TP+FN}$ (per la classe positiva). Questa metrica esprime la percentuale di istanze positive che sono state correttamente identificate rispetto al totale delle istanze che potevano essere correttamente identificate. In questo lavoro, per quanto rimanga importante avere valori vicini a 100%, non raggiungere precisamente l'obiettivo significa essere più prudenti del necessario con un paziente, che non è di per se negativo. In medicina il recupero sulla classe positiva è detto anche *sensitività*, quello sulla classe negativa è detto *specificità*.

- **F1-score** è una espressione combinata di *precisione* e *recall* che si calcola come la media armonica tra le due metriche $\frac{2*prec*recall}{prec+recall}$. Anche per questa metrica si possono fare i calcoli sulla singola classe e combinarli con la media.

A queste metriche, va aggiunga una misurazione dell'intervallo di confidenza con cui vengono fornite. Questo aiuta a definire il reale valore del modello e aiuta nel confronto tra più modelli.

L'idea dietro alla costruzione dell'intervallo di confidenza è che si possano modellare i risultati forniti dal modello come una variabile aleatoria con distribuzione binomiale $B(n, p)$ dove n è il numero di esperimenti fatti e p è la probabilità di successo. Se il calcolo delle metriche è fatto su un numero abbastanza grande di prove (almeno 30) allora la variabile binomiale può essere approssimata con una variabile aleatoria con distribuzione normale $N(np, npq)$ dove $q = 1 - p$. Questo significa che si può pensare che il risultato ottenuto con il calcolo della metrica sia al centro della curva gaussiana e che ci sia una probabilità non nulla che il valore reale della metrica (se calcolato su infiniti esempi) sia un altro. Per calcolare l'intervallo di confidenza all'interno del quale si può affermare con una determinata confidenza (diciamo del 95% che è una confidenza molto comune) che si trovi l'accuratezza reale si usa la funzione di ripartizione della distribuzione normale $P(\mu - z < p/\sigma < \mu + z) = c$. Una volta stabilità la confidenza c si può ricavare il valore di z (tramite tabelle, nel caso $c = 0.95 \rightarrow z \approx 1.96$) e risolvendo per il valore medio si ottiene l'intervallo di confidenza.

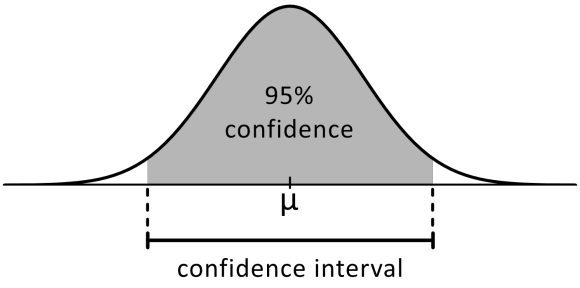

Figura 2.6: Area sotto la curva .
Fonte: omnicalculator.com

2.5 Selezione dei migliori modelli

All'inizio del lavoro, non sapendo quali modelli potessero essere i migliori, sono stati eseguiti molti test su molti modelli. Seguono i risultati (con l'ampiezza del relativo intervallo di confidenza impostato al 95%) ottenuti con le migliori combinazioni di iperparametri; ho ritenuto importante mantenere la distinzione delle metriche tra classe positiva e negativa perché in questo lavoro sono molto importanti i valori che riguardano la classe positiva (i "non gravi").

Modello	Acc.	Prec. Ps.	Prec. Ng.	Recall Ng.	Recall Ps.
Reg. Logistic	71 ± 12	69 ± 17	73 ± 16	70 ± 17	72 ± 17
SVC	78 ± 11	82 ± 14	75 ± 16	81 ± 15	76 ± 16
Rand Forest	76 ± 12	73 ± 17	79 ± 16	74 ± 16	77 ± 17
KNN	71 ± 12	55 ± 19	86 ± 12	66 ± 16	80 ± 18
XGBoost	75 ± 12	76 ± 16	75 ± 17	76 ± 17	75 ± 16
CatBoost	78 ± 11	76 ± 16	80 ± 15	77 ± 16	80 ± 16
LightGBM	65 ± 13	70 ± 16	61 ± 19	69 ± 17	64 ± 18
MLP	82 ± 10	82 ± 16	85 ± 17	85 ± 17	80 ± 16
TabNet	57 ± 12	75 ± 12	35 ± 15	53 ± 21	60 ± 15
Riferimento	86 ± 8	100	72 ± 17	100	78 ± 14

Tabella 2.2: Riassunto dei risultati ottenuti.

Per questioni di spazio ho dovuto usare alcune abbreviazioni e arrotondare i risultati.

2.6 Combinazione dei modelli

Siccome nessuno dei modelli addestrati si avvicinava al modello di riferimento, ho deciso di provare a migliorare la previsione utilizzando tutti i modelli con i risultati migliori combinandoli attraverso un multilayer perceptron (come l'idea sfruttata dello *stacking*) e di esaminare architetture sia semplici che complesse, implementandole grazie alla flessibilità offerta dalla libreria Keras [41] .

Abbreviazione	Completo
Reg. Logistica	Regressione Logistica
SVC	Support Vector Classifier
KNN	K-nearest neighbors
MLP	Multilayer Perceptron
Acc.	Accuratezza
Prec.	Precisione
Ng.	Classe Negativa
Ps.	Classe Positiva

Tabella 2.3: Abbreviazioni nei risultati.

Ho considerato che, per la struttura del problema, le metriche più rilevanti sulle quali concentrarsi fossero la precisione sulla classe positiva (perché avere troppi falsi negativi sarebbe un grosso problema) e il *recall* sulla classe positiva (in quanto migliorare questa metrica significherebbe migliorare le letture dalla TAC su stenosi lievi, che era l'obiettivo iniziale del progetto). Per cercare di estrarre queste caratteristiche dai modelli già esistenti, ho organizzato i modelli in due liste e ordinato le due liste per precisione sulla classe positiva e sensitività. Dalla classifica ordinata per precisione ho preso i migliori 3 modelli e dalla classifica ordinata per sensitività ho preso i migliori 2 modelli.

Dopo aver testato diverse alternative con diverse architetture relativamente semplici, quella che ha ottenuto i migliori risultati è stata leggermente più complessa delle altre e presentava una struttura ad "Y". Dall'alto venivano inseriti in un ramo i valori delle stenosi come presentati nel dataset, dall'altro ramo venivano inserite le previsioni effettuate dai migliori modelli (prima addestrati sullo stesso sottoinsieme di istanze usato anche per l'addestramento del multilayer perceptron). I due input sono prima elaborati indipendentemente, poi vengono concatenati nell'ultimo tratto della rete, che termina con due nodi attivati dalla funzione sigmoide.

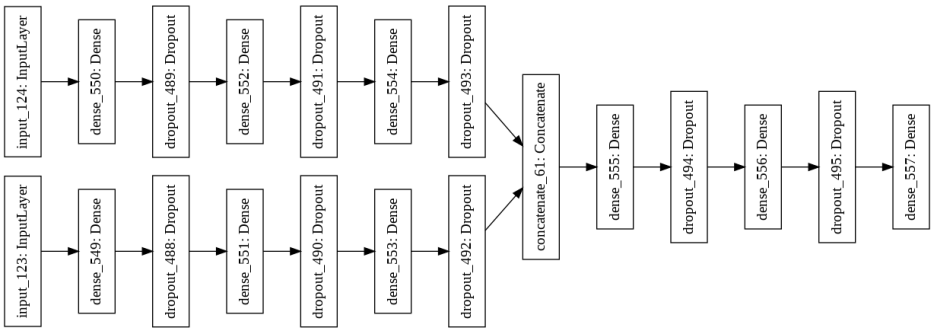

Figura 2.7: Disegno del multilayer perceptron come presentato da Keras.

Capitolo 3

Conclusioni

3.1 Confronto tra modello di riferimento e risultati finali

Riporto i risultati del modello migliore accanto a quello di riferimento.

Modello	Acc.	Prec. Ps.	Prec. Ng.	Recall Ng.	Recall Ps.
MLP Finale	85 ± 9	85 ± 11	85 ± 12	86 ± 11	86 ± 12
Riferimento	86 ± 8	100	72 ± 17	100	78 ± 14

Tabella 3.1: Riassunto dei risultati ottenuti.

Alla luce dei risultati finali, si può notare che il modello di riferimento (che ricordo, simula una valutazione molto semplificata della TAC) non ha *mai* falsi positivi, che significa che non ci sono persone valutate come "non gravi" dopo una TAC e che venivano rivalutati come "gravi" dopo la coronarografia. Nel corso di questo lavoro non è stato possibile raggiungere ne la stessa accuratezza ne la stessa precisione del modello di riferimento, ma si può notare come il recupero sulla classe positiva, che indica quante tra le persone con stenosi non gravi sono state correttamente identificate, sia molto migliorato. Questo miglioramento può essere notato anche osservando la percentuale di persone che sono classificate come non gravi (indipendentemente dalla reale classificazione) dai due modelli, 50% per il multilayer perceptron e 36% dal modello di riferimento.

Questo miglioramento viene però ad un costo, il costo dei falsi positivi. Considerando una popolazione di 100 pazienti, 50 di essi verranno considerati

tra quelli con le stenosi lievi e 8 di questi sarebbero però state persone che avrebbero necessitato di aiuto.

3.2 Possibili applicazioni

Il software prodotto in questo lavoro è lontano dal poter essere distribuito ed utilizzato realmente in un ospedale.

Però si potrebbe incominciare a verificare l'effettiva utilità del modello testandolo sul campo, senza che possa avere effetti reali sulle decisioni prese. A seguito di questo periodo di test, nel caso i test verificassero l'effettiva utilità del modello di previsione, questo potrebbe essere inserito all'interno dello spettro di informazioni disponibili ad un cardiologo al momento della definizione della prognosi. Non credo ci sia bisogno di spiegare i vantaggi (ed i problemi che si evitano) lasciando un umano alla fine della catena decisionale.

3.3 Possibili sviluppi futuri

I risultati ottenuti, lasciano pensare che ci sia spazio per ulteriori miglioramenti e alcune strade che si possono percorrere per raggiungere questi miglioramenti sono:

- Raccogliere più dati dello stesso tipo. Questo dataset aveva 165 letture, che sono poche se comparate ai dataset tipici (si possono andare a vedere i dataset messi a disposizione da Tensorflow [42] o Kaggle).
- Raccogliere dati sempre relativi alle TAC, ma nella forma grezza di immagini tridimensionali. Aumenterebbe enormemente la dimensionalità e la quantità di dati, necessiterebbe di molto lavoro a monte delle previsioni, ma si manterrebbero molti dettagli persi durante la codifica secondo le linee guida SCCT.
- Raccogliere dati con la tecnica quantitativa (ovvero con le stenosi valutate da un software) per avere dei dati più variegati con più dettagli e dimensioni che possono essere gestiti da tecniche di apprendimento automatico.
- Entrare nel dettaglio della pratica clinica per osservare un quadro più ampio ed analizzare se i falsi positivi vengono filtrati ed esclusi tramite qualche altro esame funzionale effettuato precedentemente.

Ringraziamenti

Il primo ringraziamento va al relatore di questo lavoro, il Prof. Gianluca Moro, che mi ha aiutato in questi mesi e ha reso possibile tutto questo.

Un ringraziamento all'ospedale Villa Maria, in particolare al direttore generale ed amministratore delegato Dott. Lorenzo Venturini per aver concesso l'utilizzo dei dati, al Dott. Ilja Gardi e al Dott. Massimo Margheri che hanno organizzato lo studio che ha prodotto i dati.

Grazie a Ing. Luca Ghetti che mi ha messo in collegamento con l'ospedale e che in questi ultimi anni mi ha dato l'opportunità di crescere in svariati aspetti della mia vita.

Gli ultimi ringraziamenti vanno alla mia famiglia perché mi ha permesso di proseguire questi anni di studi senza preoccupazioni, agli amici perché ogni volta che si faceva dura mi hanno ricaricato di energia e Virgilia per avermi accompagnato.

Bibliografia

[1] Kevin D. Hill and Andrew J. Einstein. New approaches to reduce radiation exposure. *Trends in Cardiovascular Medicine*, 26(1):55–65, January 2016.

[2] Franz-Josef Neumann, Miguel Sousa-Uva, Anders Ahlsson, Fernando Alfonso, Adrian P Banning, Umberto Benedetto, Robert A Byrne, Jean-Philippe Collet, Volkmar Falk, Stuart J Head, Peter Jüni, Adnan Kastrati, Akos Koller, Steen D Kristensen, Josef Niebauer, Dimitrios J Richter, Petar M Seferović, Dirk Sibbing, Giulio G Stefanini, Stephan Windecker, Rashmi Yadav, Michael O Zembala, and ESC Scientific Document Group. 2018 ESC/EACTS Guidelines on myocardial revascularization. *European Heart Journal*, 40(2):87–165, 08 2018.

[3] SCOT-HEART investigators. Ct coronary angiography in patients with suspected angina due to coronary heart disease (scot-heart): an open-label, parallel-group, multicentre trial. *Lancet (London, England)*, 385(9985):2383—2391, June 2015.

[4] J Ronald Mikolich. Cardiac computed tomographic angiography and the primary care physician. *The Journal of the American Osteopathic Association*, 112(5):267—275, May 2012.

[5] Gurpreet Singh, Subhi J. Al'Aref, Marly Van Assen, Timothy Suyong Kim, Alexander van Rosendael, Kranthi K. Kolli, Aeshita Dwivedi, Gabriel Maliakal, Mohit Pandey, Jing Wang, Virginie Do, Manasa Gummalla, Carlo N. De Cecco, and James K. Min. Machine learning in cardiac ct: Basic concepts and contemporary data. *Journal of Cardiovascular Computed Tomography*, 12(3):192–201, 2018.

[6] Su Yang, Jihoon Kweon, J. Roh, Jae-Hwan Lee, H. Kang, Lae-Jeong Park, D. J. Kim, Hyeonkyeong Yang, Jaehee Hur, Do-Yoon Kang, P. Lee, Jung-Min Ahn, Soo-Jin Kang, Duk-Woo Park, Seung-Whan Lee, Young-Hak

Kim, C. Lee, S. Park, and Seung-Jung Park. Deep learning segmentation of major vessels in x-ray coronary angiography. *Scientific Reports*, 9, 2019.

[7] Tianming Du, Lihua Xie, Honggang Zhang, Xuqing Liu, Xiaofei Wang, Donghao Chen, Yang Xu, Zhongwei Sun, Wenhui Zhou, Lei Song, Changdong Guan, Alexandra J Lansky, and Bo Xu. Training and validation of a deep learning architecture for the automatic analysis of coronary angiography. *EuroIntervention : journal of EuroPCR in collaboration with the Working Group on Interventional Cardiology of the European Society of Cardiology*, 17(1):32—40, May 2021.

[8] Hyungjoo Cho, June-Goo Lee, Soo-Jin Kang, Won-Jang Kim, So-Yeon Choi, Jiyuon Ko, H. Min, Gunho Choi, Do-Yoon Kang, P. Lee, Jung-Min Ahn, Duk-Woo Park, Seung-Whan Lee, Young-Hak Kim, C. Lee, Seong-Wook Park, and Seung-Jung Park. Angiography-based machine learning for predicting fractional flow reserve in intermediate coronary artery lesions. *Journal of the American Heart Association: Cardiovascular and Cerebrovascular Disease*, 8, 2019.

[9] Alan C. Kwan, Priscilla A. McElhinney, Balaji K. Tamarappoo, Sebastien Cadet, Cecilia Hurtado, Robert J. H. Miller, Donghee Han, Yuka Otaki, Evann Eisenberg, Joseph E. Ebinger, Piotr J. Slomka, Victor Y. Cheng, Daniel S. Berman, and Damini Dey. Prediction of revascularization by coronary CT angiography using a machine learning ischemia risk score. *European Radiology*, 31(3):1227–1235, September 2020.

[10] Verena Brandt, Tilman Emrich, U. Joseph Schoepf, Danielle M. Dargis, Richard R. Bayer, Carlo N. De Cecco, and Christian Tesche. Ischemia and outcome prediction by cardiac CT based machine learning. *The International Journal of Cardiovascular Imaging*, 36(12):2429–2439, July 2020.

[11] Taner Sen, Celal Kilit, Mehmet Ali Astarcioglu, Lale Dinc Asarcikli, Tolga Aksu, Habibe Kafes, Afsin Parspur, Gokhan Gozubuyuk, and Basri Amasyali. Comparison of quantitative and qualitative coronary angiography: computer versus the eye. *Cardiovascular Journal of Africa*, 29(5):278–282, October 2018.

[12] Jonathon Leipsic, Suhny Abbara, Stephan Achenbach, Ricardo Cury, James P. Earls, GB John Mancini, Koen Nieman, Gianluca Pontone, and

Gilbert L. Raff. SCCT guidelines for the interpretation and reporting of coronary CT angiography: A report of the society of cardiovascular computed tomography guidelines committee. *Journal of Cardiovascular Computed Tomography*, 8(5):342–358, September 2014.

[13] The pandas development team. pandas-dev/pandas: Pandas, February 2020.

[14] Raymond J. Gibbons, Kanu Chatterjee, Jennifer Daley, John S. Douglas, Stephan D. Fihn, Julius M. Gardin, Mark A. Grunwald, Daniel Levy, Bruce W. Lytle, Robert A. O'Rourke, William P. Schafer, Sankey V. Williams, James L. Ritchie, Raymond J. Gibbons, Melvin D. Cheitlin, Kim A. Eagle, Timothy J. Gardner, Arthur Garson, Richard O. Russell, Thomas J. Ryan, and Sidney C. Smith. ACC/AHA/ACP–ASIM guidelines for the management of patients with chronic stable angina: Executive summary and recommendations. *Circulation*, 99(21):2829–2848, June 1999.

[15] J.S. Cramer. The Origins of Logistic Regression. Tinbergen Institute Discussion Papers 02-119/4, Tinbergen Institute, December 2002.

[16] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory - COLT '92*. ACM Press, 1992.

[17] Evelyn Fix and J. L. Hodges. Discriminatory analysis. nonparametric discrimination: Consistency properties. *International Statistical Review / Revue Internationale de Statistique*, 57(3):238–247, 1989.

[18] Paul E. Utgoff. Incremental induction of decision trees. *Mach. Learn.*, 4:161–186, 1989.

[19] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[20] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20(8):832–844, 1998.

[21] Tin Kam Ho. Random decision forests. In *Third International Conference on Document Analysis and Recognition, ICDAR 1995, August 14 - 15, 1995, Montreal, Canada. Volume I*, pages 278–282. IEEE Computer Society, 1995.

[22] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189 – 1232, 2001.

[23] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[24] Anna Veronika Dorogush, Vasily Ershov, and Andrey Gulin. Catboost: gradient boosting with categorical features support. *CoRR*, abs/1810.11363, 2018.

[25] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 3146–3154, 2017.

[26] Fionn Murtagh. Multilayer perceptrons for classification and regression. *Neurocomputing*, 2(5):183–197, 1990.

[27] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6:386–408, 1958.

[28] Sercan Ömer Arik and Tomas Pfister. Tabnet: Attentive interpretable tabular learning. *CoRR*, abs/1908.07442, 2019.

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[30] Antonio D'Ambrosio and Valerio A. Tutore. Conditional classification trees by weighting the gini impurity measure. In *Studies in Classification, Data Analysis, and Knowledge Organization*, pages 273–280. Springer Berlin Heidelberg, 2011.

[31] Léon Bottou. Online algorithms and stochastic approximations. In David Saad, editor, *Online Learning and Neural Networks*. Cambridge University Press, Cambridge, UK, 1998. revised, oct 2012.

[32] D. Rumelhart, Geoffrey E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[33] M. N. Murty and Rashmi Raghava. Kernel-based SVM. In *Support Vector Machines and Perceptrons*, pages 57–67. Springer International Publishing, 2016.

[34] Martin D. Buhmann. *Radial Basis Functions - Theory and Implementations*, volume 12 of *Cambridge monographs on applied and computational mathematics*. Cambridge University Press, 2009.

[35] Stephen M. Omohundro. Five balltree construction algorithms. Technical report, 1989.

[36] Mouhammd Alkasassbeh, Ghada Awad Altarawneh, and Ahmad B. A. Hassanat. On enhancing the performance of nearest neighbour classifiers using hassanat distance metric. *CoRR*, abs/1501.00687, 2015.

[37] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001.

[38] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[39] Alex Sherstinsky. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *CoRR*, abs/1808.03314, 2018.

[40] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey J. Gordon, David B. Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011*, volume 15 of *JMLR Proceedings*, pages 315–323. JMLR.org, 2011.

[41] Francois Chollet et al. Keras, 2015.

[42] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving,

Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

Codice

Si aggiunge in appendice il codice prodotto.

```
# -*- coding: utf-8 -*-
"""
## Download dati
e altre variabili
"""

from google_drive_downloader import GoogleDriveDownloader as gdd
from os import remove, path


# sono i modelli per i quali ho preparato la GridSearch
all_models = ["svm_reg", "tree_reg", "knn_reg", "xgb_reg", "catbst_reg", "lgbm_reg",
                    "svm_cls", "tree_cls", "knn_cls", "xgb_cls", "catbst_cls", "lgbm_cls", "logreg_cls"
                    ]
usually_best_models = ["tree_reg", "xgb_reg", "knn_reg",
                    "tree_cls", "xgb_cls", "knn_cls"
                    ]


# aumentando 'regolarzn' con valori tra 0 e 2 aumenta la regolarizzazione
# dei vari modelli nelle pipeline
regolarzn = 2
# 'available_models' è un riassunto di 'all_models' che serve a fare risparmiare
# un po di tempo nell' addestramento
available_models = all_models
# numero di prove da fare con i modelli di sklearn
num_experiments_skl_models = 5
# rimuovo eventuali vecchi file
data_file_path = "data/dataset.xlsx"
if path.exists(data_file_path):
```

```
31    remove("/content/" + data_file_path )
32
33  # scarico da drive il file excell con le valutazioni delle lesioni
34  gdd.download_file_from_google_drive(
35      file_id='1RbbBUQycR44bq8ROqlxAskDP4lwSN1_a',
36      dest_path=data_file_path
37      )
38
39  """## Import vari"""
40
41
42  from sklearn.preprocessing import StandardScaler, PolynomialFeatures,
    ↪  MinMaxScaler
43  from sklearn.model_selection import train_test_split, StratifiedKFold,
    ↪  KFold, GridSearchCV, cross_val_score, cross_validate
44  from sklearn.metrics import roc_curve, precision_recall_curve, auc
45  from sklearn.metrics import mean_squared_error, mean_absolute_error
46  from sklearn.exceptions import ConvergenceWarning
47
48  from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier
49  from sklearn.svm import SVR, SVC
50  from sklearn.linear_model import Perceptron, LogisticRegression
51  from sklearn.neighbors import KNeighborsRegressor, KNeighborsClassifier
52
53  from imblearn.pipeline import Pipeline
54  from imblearn.over_sampling import SMOTE, SVMSMOTE
55  from imblearn.under_sampling import InstanceHardnessThreshold
56
57  from catboost import CatBoostClassifier, CatBoostRegressor
58  import xgboost as xgb
59  from lightgbm import LGBMRegressor, LGBMClassifier
60
61  from mlxtend.classifier import EnsembleVoteClassifier
62
63  import random
64  from functools import reduce
65
66  import pandas as pd
67  from matplotlib import pyplot as plt
68  import numpy as np
69  np.warnings.filterwarnings('ignore', category=np.VisibleDeprecationWarning)
70  np.warnings.filterwarnings('ignore', category=RuntimeWarning) # DA FARE
    ↪  ATTENZIONE
71  import warnings
72  warnings.simplefilter(action='ignore', category=FutureWarning)
```

```
warnings.simplefilter(action='ignore', category=ConvergenceWarning)

"""## Funzioni per preparare i dati

### Caricamento dati
"""

def load_xcell_in_dataframe(get_only_pairs = True):
  # carico i dati dal file excell dentro a 3 dataframe
  # divisi per 'operatore' che ha fatto la valutazione e l'esame che è stato valutato
  coronarografia_op1_full = pd.read_excel("/content/data/dataset.xlsx", "Foglio1", usecols = "B:X" )
  coronarografia_op2_full = pd.read_excel("/content/data/dataset.xlsx", "Foglio1", usecols = "Z:AV", names = coronarografia_op1_full.columns)
  taccoronarica_op1_full = pd.read_excel("/content/data/dataset.xlsx", "Foglio1", usecols = "AX:BT", names = coronarografia_op1_full.columns)
  taccoronarica_op2_full = pd.read_excel("/content/data/dataset.xlsx", "Foglio1", usecols = "BV:CR", names = coronarografia_op1_full.columns)

  # ci sono alcuni dati che mancano perche alcuni pazienti hanno
  # fatto la tac per valutare 'il calcio' e non è possibile osservare le stenosi in questi casi
  coro_op1 = coronarografia_op1_full.convert_dtypes(convert_integer=True)
  coro_op2 = coronarografia_op2_full.convert_dtypes(convert_integer=True)
  tac_op1 = taccoronarica_op1_full.convert_dtypes(convert_integer=True)
  tac_op2 = taccoronarica_op2_full.convert_dtypes(convert_integer=True)

  # check correttezza
  assert coro_op1.shape == coro_op2.shape and coro_op2.shape == tac_op2.shape and tac_op2.shape == tac_op1.shape


  if not get_only_pairs:
    return tac_op1, tac_op2, coro_op1, coro_op2
  else:
    # calcolo una lista di tutti gli indici con dei buchi, per poterli filtrare
    indices_with_na = set(list(reduce(
      lambda lista, indici: lista + list(indici),
      map(
          lambda df:df[df.isna().any(axis=1)].index,
          [coro_op1, coro_op2, tac_op1, tac_op2]
```

```
108         ),
109         []
110     )))
111 
112     indices_without_na = set(list(taccoronarica_op2_full.index)) -
    ↪  indices_with_na
113 
114     # filtro i dati per i pazienti(righe) che sono presenti in tutti e tre
    ↪  gli operatori
115     coro_op1 = coro_op1[coro_op1.index.isin(indices_without_na)]
116     coro_op2 = coro_op2[coro_op2.index.isin(indices_without_na)]
117     tac_op1  = tac_op1[ tac_op1.index.isin(indices_without_na)]
118     tac_op2  = tac_op2[ tac_op2.index.isin(indices_without_na)]
119 
120 
121     return tac_op1, tac_op2, coro_op1, coro_op2
122 
123 """
124 ### Preparazione Dati"""
125 
126 def get_x_y_from_dataset(tac_op2, coro_op1, coro_op2):
127   X = np.array([[0]*tac_op2.shape[1]])
128   y = np.array([[0]*coro_op1.shape[1]])
129   for i in range( coro_op1.shape[0]):
130     c1 = np.array(coro_op1.iloc[i])
131     c2 = np.array(coro_op2.iloc[i])
132     t2 = np.array(tac_op2.iloc[i])
133 
134     X = np.append(X,np.array([t2]), axis = 0)
135     y = np.append(y,np.array([c1]), axis = 0)
136 
137   return X[1:], y[1:]
138 
139 def prepare_data_for_model(tac_op2, coro_op1, coro_op2,
140                            column_filter:list = None,
141                            model_name:str="tree_cls",
142                            usa_trucchetto = False,
143                            cosa_cercare:int = 0):
144   """
145   sostanzialmente ho deciso di predirre se con una coronarografia
    ↪  (invasiva) i pazienti sarebbero mandati all operazione
146   se riesco a capire da una tac che la coronarografia non li manderebbe
    ↪  all operazione, posso risparmiare alle persone la coronarografia
147 
```

```
148   una stenosi è do perare quando l'otturazione è superiore al 50%, nel
      ↪  caso del mio dataset
149   le stenosi sono categorizzate da 1 a 6:
150   1 -> 0% (tutto ok)
151   4 -> 50-69%
152   6 -> 100% (ischemia)
153
154   cosa_cercare può essere:
155   0 - normale classificazione
156   1 - cerca i falsi negativi
157   2 - cerca quelle istanze che sono identiche ma che vengono classificate
      ↪  diversamente
158   """
159   assert model_name in available_models
160
161
162   if not column_filter:
163     X,y = get_x_y_from_dataset(tac_op2, coro_op1, coro_op2, usa_trucchetto)
164   else:
165     X,y = get_x_y_from_dataset(
166         tac_op2[column_filter],
167         coro_op1[column_filter],
168         coro_op2[column_filter],
169         usa_trucchetto)
170
171   _y = []
172   for t,c in zip(X, y):
173     t_operare = t[0] >= 4 or any(t >= 5)
174     c_operare = c[0] >= 4 or any(c >= 5)
175
176
177     condizione_classificazione = {
178         0: not t_operare or not c_operare,
179         1: t_operare and not c_operare,
180         2: (t_operare != c_operare) and (np.all((t - c) == 0)),
181         3: t_operare
182     }[cosa_cercare]
183
184
185     if condizione_classificazione:
186       _y.append( 1 )
187     else:
188       _y.append( -1 )
189
190   if model_name == "xgboost":
```

```
191     _y = convert_y_for_xgboost(_y)
192
193   return X.astype(np.int), np.array(_y).astype(np.int)
194
195
196 def convert_y_for_xgboost(y):
197   return np.array(
198       list(
199           map( lambda yi:0 if yi == -1 else 1, y )
200       )
201   )
202
203 def plot_data(X, y):
204   """
205   stampa qualche informazione basilare su i dati
206   """
207
208   y = convert_y_for_xgboost(y)
209   pd.DataFrame(np.bincount(X.ravel())).plot.bar(
210       title = "Distribuzione delle gravita delle stenosi"
211     )
212
213   pd.DataFrame(np.bincount(y.ravel())).plot.bar(
214       title = "Distribuzione della necessita di operazioni",
215       xlabel = "1 = da NON operare"
216     )
217
218 def split_data(X,y, test_size = 1/4):
219   """
220   separa i dati in train e test (nel caso si addestri 'xgboost' imposta
    ↪  l'etichetta delle classi correttamente)
221   inoltre, ritorna anche i delle _semplici_ predizioni fatte solo sulla
    ↪  base della TAC, per simulare le decisioni COME sono prese attualmente
222   """
223
224   X_train, X_test, y_train, y_test = train_test_split(X, y,
    ↪  test_size=test_size, shuffle=True, stratify=y)
225   X_train, X_test, y_train, y_test = np.array(X_train), np.array(X_test),
    ↪  np.array(y_train).ravel(), np.array(y_test).ravel()
226   return (X_train, X_test),(y_train, y_test)
227
228 def get_split_data(test_size = 1/4, trucchetto = False, filtro_colonne =
    ↪  True):
229   """
230   Raggruppa un po tutte le funzioni dichiarate sopra
```

```
231   per ottenere in una chiamata sola X e Y gia divisi per test e train
232   """
233   relevant_columns = ['TC',
234                       'IVA_PROX', 'IVA_MEDIA',
235                       'CDX_PROX', 'CDX_MEDIO',
236                       'CFX_PROX', 'CFX_MEDIA', 'I_MARGINALE'
237                       ]
238
239   tac_op1, tac_op2, coro_op1, coro_op2 = load_xcell_in_dataframe()
240
241
242   # trasformo nel formato che mi interessa (con le classi)
243   X, y = prepare_data_for_model(
244       tac_op1, coro_op1, coro_op2,
245       column_filter=relevant_columns if filtro_colonne else None ,
246       usa_trucchetto=trucchetto
247   )
248   # divido TUTTE le vole i dati in maniera random
249   Xs, ys = split_data(X,y, test_size)
250   return Xs, ys
251
252 """## Funzioni per addestrare il modello
253
254 ### Griglie e Pipeline
255 """
256
257 # Qui c'è solo la dichiarazione di tutte le grid search
258 # ogni funzione è nominata come
    ↪  "get_<modello>_<obiettivo>_pipeline_and_grid"
259
260 def get_svm_reg_pipeline_and_grid():
261   model = Pipeline([
262     ("resampl1", None),
263     ("preproc1", None),
264     ("preproc2", None), #  = 2)
265     ("regressor", SVR()),
266   ])
267
268   grid =[
269     {
270       "preproc1": [MinMaxScaler(), StandardScaler()],
271       "regressor__C": [0.01, 0.1, 1, 10],
272       "regressor__kernel" : ["rbf", "sigmoid"]
273     },
274     {
```

```
275         "resampl1": [SVMSMOTE()],
276         "preproc1": [MinMaxScaler(), StandardScaler()],
277         "regressor__kernel": ['poly'],
278         "regressor__degree": [2, 3],
279         "regressor__C": [0.1, 1, 10 ]
280       }
281     ]
282     return model, grid
283
284   def get_tree_reg_pipeline_and_grid():
285     model = Pipeline([
286       ("resampl1", None),
287       ("preproc1", None),
288       ("preproc2", None),
289       ("regressor", RandomForestRegressor()),
290     ])
291
292     grid =[
293        {
294         "resampl1": [None, SVMSMOTE() ], # 3
295         "preproc1": [MinMaxScaler(), StandardScaler()],
296         "preproc2": [None, PolynomialFeatures(degree = 2)],
297         "regressor__max_depth" : [25, 20, 15, 10, 7,
          ↪  4][regolarzn:regolarzn+2],
298         "regressor__max_features": ["sqrt"]
299       }]
300     return model, grid
301
302   def get_svm_clss_pipeline_and_grid():
303     model = Pipeline([
304       ("resampl1", None),
305       ("preproc1", None),
306       ("preproc2", None), # (= 2)
307       ("regressor", SVC()),
308     ])
309
310     grid =[
311       {
312         "preproc1": [MinMaxScaler(), StandardScaler()],
313         "regressor__C": [0.01, 0.5, 1, 10],
314         "regressor__kernel" : ["rbf"]
315       },
316       {
317         "resampl1": [SVMSMOTE()],
318         "preproc1": [MinMaxScaler(), StandardScaler()],
```

```
319         "regressor__kernel": ['poly'],
320         "regressor__degree": [2, 3, 4],
321         "regressor__C": [10, 1, 0.1, 0.01, 0.005][regolarzn:regolarzn+2]
322       }
323     ]
324     return model, grid
325
326   def get_knn_reg_pipeline_and_grid():
327     model = Pipeline([
328       ("resampl1", None),
329       ("preproc1", None),
330       ("preproc2", None), # (= 2)
331       ("regressor", KNeighborsRegressor()),
332     ])
333
334     grid =[
335       {
336         "resampl1": [None, SVMSMOTE() ],
337         "preproc1": [None, StandardScaler()],
338         "preproc2": [None, PolynomialFeatures(degree = 2)],
339         "regressor__n_neighbors": [4, 6, 8, 10, 12,
          ↪  14][regolarzn:regolarzn+2],
340         "regressor__weights":["distance"],
341         "regressor__algorithm":["auto", "ball_tree", "brute"],
342         "regressor__p":[2, 3]
343       }
344     ]
345     return model, grid
346
347   def get_tree_clss_pipeline_and_grid():
348     model = Pipeline([
349       ("resampl1", None),
350       ("preproc1", None),
351       ("preproc2", None),
352       ("regressor", RandomForestClassifier()),
353     ])
354
355     grid =[
356       {
357         "resampl1": [None, SVMSMOTE() ], # 3
358         "preproc1": [MinMaxScaler(), StandardScaler()],
359         "preproc2": [None, PolynomialFeatures(degree = 2)],
360         "regressor__max_depth" : [25, 20, 15, 10, 7,
          ↪  4][regolarzn:regolarzn+2],
361         "regressor__max_features": ["sqrt"],
```

```
362         "regressor__ccp_alpha": [0.0, 0.2]
363       }]
364     return model, grid
365 
366 
367   def get_knn_clss_pipeline_and_grid():
368     model = Pipeline([
369       ("resampl1", None),
370       ("preproc1", None),
371       ("preproc2", None),
372       ("regressor", KNeighborsClassifier()),
373     ])
374 
375     grid =[
376       {
377         "resampl1": [None, SVMSMOTE() ], # 3
378         "preproc1": [None, StandardScaler()],         # 3
379         "preproc2": [None, PolynomialFeatures(degree = 2)],          # 2
380         "regressor__n_neighbors": [4, 6, 8, 10, 12,
          ↪  14][regolarzn:regolarzn+2],
381         "regressor__weights":["distance"],
382         "regressor__algorithm":["auto", "ball_tree", "brute"],
383         "regressor__p":[3]
384       }
385     ]
386     return model, grid
387 
388   def get_xgboost_clss_pipeline_and_grid():
389     model = Pipeline([
390       ("preproc", None),
391       ("classification", xgb.XGBClassifier(objective ='binary:logistic',
        ↪  eval_metric='logloss', use_label_encoder=False) )
392     ])
393 
394     grid = [{
395       "preproc":[MinMaxScaler(), StandardScaler()],
396       "classification__colsample_bytree": [1],
397       "classification__learning_rate": [0.45, 0.65],
398       "classification__max_depth": [25, 20, 15, 10][regolarzn:regolarzn+2],
399       "classification__n_estimators": [10, 20],
400       "classification__alpha":  [ 1, 10, 50, 100,
        ↪  500][regolarzn:regolarzn+2],
401       "classification__lambda": [ 1, 10, 50, 100, 500][regolarzn:regolarzn+2]
402     }]
403     return model, grid
```

```
404
405 def get_xgboost_reg_pipeline_and_grid():
406   model = Pipeline([
407     ("preproc", None),
408     ("preproc2",None),
409     ("classification", xgb.XGBRegressor(objective ='reg:squarederror') )
410   ])
411
412   grid = [{
413     "preproc":[MinMaxScaler(), StandardScaler()],
414     "preproc2": [None, PolynomialFeatures(degree=2)],
415
416     "classification__colsample_bytree": [1],
417     "classification__learning_rate": [0.45, 0.65],
418     "classification__max_depth": [25, 20, 15, 10, 7,
      ↪  4][regolarzn:regolarzn+2],
419     "classification__n_estimators": [10, 20],
420     "classification__alpha": [ 1, 10, 50, 100, 500][regolarzn:regolarzn+2],
421     "classification__lambda": [ 1, 10, 50, 100, 500][regolarzn:regolarzn+2]
422   }]
423   return model, grid
424
425 def get_catboost_cls_pipeline_and_grid():
426   model = Pipeline([
427     ("preproc", None),
428     ("classification", CatBoostClassifier(verbose=0))
429   ])
430
431   grid = [{
432     "preproc":[MinMaxScaler(), StandardScaler()],
433     "classification__iterations": [2,5,8],
434     "classification__learning_rate": [0.1, 0.5, 1],
435     "classification__depth": [16, 13, 10, 7, 5][regolarzn:regolarzn+2],
436   }]
437   return model, grid
438
439 def get_catboost_reg_pipeline_and_grid():
440   model = Pipeline([
441     ("preproc", None),
442     ("classification", CatBoostRegressor(verbose=0))
443   ])
444
445   grid = [{
446     "preproc":[MinMaxScaler(), StandardScaler()],
447     "classification__iterations": [2,5,8],
```

```
448       "classification__learning_rate": [0.1, 0.5, 1],
449       "classification__depth": [16, 13, 10, 7, 5][regolarzn:regolarzn+2],
450     }]
451     return model, grid
452 
453 def get_logreg_cls_pipeline_and_grid():
454     p = Pipeline([
455         ("preproc1", None),
456         ("preproc2", None),
457         ("regressor", LogisticRegression(
458             solver="saga",
459             max_iter = 1500,
460             penalty = "elasticnet"
461             )
462         )
463     ])
464 
465     grid =[
466         {
467             "preproc1": [StandardScaler(), MinMaxScaler()],
468             "preproc2": [None, PolynomialFeatures(degree=2)],
469             "regressor__C": [0.1, 1, 10],
470             "regressor__l1_ratio":[0.1, 0.3, 0.5, 0.7, 0.9]
471         }
472     ]# > 8 prove
473     return p, grid
474 
475 def get_lgbm_reg_pipeline_and_grid():
476     p = Pipeline([
477         ("preproc1", None),
478         ("preproc2", None),
479         ("regressor", LGBMRegressor() )
480     ])
481 
482     grid =[
483         {
484           "preproc1": [StandardScaler(), MinMaxScaler()],
485           "preproc2": [None, PolynomialFeatures(degree=2)],
486           "regressor__num_leaves": [5, 10, 15],
487           "regressor__learning_rate ":[0.1, 0.5, 0.9],
488           "regressor__n_estimators":[5, 10],
489           "regressor__lambda_l1": [0.1, 1, 10, 50,
              ↪  100][regolarzn:regolarzn+2],
490           "regressor__lambda_l2": [0.1, 1, 10, 50,
              ↪  100][regolarzn:regolarzn+2],
```

```
491           "regressor__min_gain_to_split": [0.5, 1, 10]
492       }
493   ]# > 8 prove
494   return p, grid
495
496 def get_lgbm_cls_pipeline_and_grid():
497   p = Pipeline([
498       ("preproc1", None),
499       ("preproc2", None),
500       ("regressor", LGBMClassifier() )
501   ])
502
503   grid =[
504       {
505         "preproc1": [StandardScaler(), MinMaxScaler()],
506         "preproc2": [None, PolynomialFeatures(degree=2)],
507         "regressor__num_leaves": [5, 10, 22],
508         "regressor__learning_rate ":[0.1, 0.5, 0.9],
509         "regressor__n_estimators":[5, 10],
510         "regressor__lambda_l1": [0.1, 1, 10, 50,
          ↪  100][regolarzn:regolarzn+2],
511         "regressor__lambda_l2": [0.1, 1, 10, 50,
          ↪  100][regolarzn:regolarzn+2],
512         "regressor__min_gain_to_split": [0.5, 1, 10]
513       }
514   ]# > 8 prove
515   return p, grid
516
517 """### Altri modelli e addestramento"""
518
519 def train_model_with_sklearn(X_train, y_train, model_name:str, verbosity =
  ↪  8):
520   """
521   addestra un modello con la gridsearch di sklearn
522   """
523   pipeline, grid = {
524     "svm_reg":    get_svm_reg_pipeline_and_grid(),
525     "svm_cls":    get_svm_clss_pipeline_and_grid(),
526     "tree_reg":   get_tree_reg_pipeline_and_grid(),
527     "knn_reg":    get_knn_reg_pipeline_and_grid(),
528     "tree_cls":   get_tree_clss_pipeline_and_grid(),
529     "knn_cls":    get_knn_clss_pipeline_and_grid(),
530     "xgb_cls":    get_xgboost_clss_pipeline_and_grid(),
531     "xgb_reg":    get_xgboost_reg_pipeline_and_grid(),
532     "catbst_cls": get_catboost_cls_pipeline_and_grid(),
```

```
533       "catbst_reg": get_catboost_reg_pipeline_and_grid(),
534       "logreg_cls": get_logreg_cls_pipeline_and_grid(),
535       "lgbm_reg":   get_lgbm_reg_pipeline_and_grid(),
536       "lgbm_cls":   get_lgbm_cls_pipeline_and_grid()
537     }[model_name]
538
539
540     grid_search = GridSearchCV(
541       pipeline,
542       grid,
543       cv = StratifiedKFold(3, shuffle=True,
          ↪  random_state=int(np.random.random() * 100)), # i dati di train
          ↪  sono ulteriormente splittati
544       n_jobs = 4,
545       verbose = verbosity
546     )
547     grid_search.fit(X_train, y_train)
548     model = grid_search.best_estimator_
549
550     return model, grid_search
551
552   class NaiveTACRegressor:
553     """
554     È un classificatore compatibile con SKLearn
555     che uso come -riferimento-
556     che si comporta "come un dottore" (piu o meno) che decide se operare o
      ↪  meno
557     """
558     def __init__(self, negative_label:int=-1 ):
559       self.nl = negative_label
560       self.gravita_stenosi_da_operare = 4
561       self.pl = 1
562
563     def fit(self, X,y):
564       return self
565
566     def predict(self, X):
567       da_operare = lambda xi: (xi[0] >= self.gravita_stenosi_da_operare) or
          ↪  any(xi >= (self.gravita_stenosi_da_operare +1))
568       results = map(
569           lambda xi: self.nl if da_operare(xi) else self.pl,
570           X
571       )
572       return np.array(list(results))
573
```

```
574   def score(self, X, y):
575     y_pred = self.predict(X)
576     u = ((y - y_pred) ** 2).sum()
577     v = ((y - y.mean()) ** 2).sum()
578     return 1 - u/v
579 
580   def predict_proba(self, X):
581     return self.predict(X)
582 
583 class NaiveTACClassifier(NaiveTACRegressor):
584   def score(self, X, y):
585     raise Warning("non chiamarmi")
586 
587 def train_model(X_train, y_train, model_name:str, verbosity = 8 ):
588   assert model_name in available_models
589 
590   if not "keras" in model_name:
591     return train_model_with_sklearn(X_train, y_train, model_name,
      ↪  verbosity)
592   else:
593     """
594     builda un modello con keras, compila e addestra
595     """
596     pass
597 
598 def print_grid_search_result(grid_search, qt = 3):
599   """
600   serve a stampare sintenticamente i risultati della gridsearch
601   """
602   interesting_columns = ["mean_fit_time", "params", "mean_test_score"]
603   result = pd.DataFrame(grid_search.cv_results_)
604 
605   print(
606       result.sort_values("rank_test_score",
      ↪  ascending=True)[interesting_columns].head(qt)
607       )
608 
609 """## Funzioni per valutare il modello"""
610 
611 from sklearn.metrics import confusion_matrix, accuracy_score
612 
613 def get_confusion_matrix(y_true, y_pred):
614   return nicer_confusion_matrix(
615       confusion_matrix(y_true, y_pred)
616   )
```

```
617
618 def nicer_confusion_matrix(cm):
619   """
620   inserisce la matrice di confusione in un dataframe con le colonne
    ↪  nominate
621   """
622   real_multi_index = pd.MultiIndex.from_product([    ['real'],    ['lett
    ↪  grav','lett non grave']  ])
623   predicted_multi_index = pd.MultiIndex.from_product([ ['predicted'],
    ↪  ['lett grav','lett non grave'] ])
624   return pd.DataFrame(data = cm, columns = predicted_multi_index, index =
    ↪  real_multi_index )
625
626 def get_scores_from_confusion_matrix(confusion_matrix: pd.DataFrame,
    ↪  avg:bool = False):
627   """
628   potevo importare precision_score, recall_score, accuracy_score, f1_score
629   ma era troppo lento, con questa funzione risparmio un po di tempo
630   """
631   # [predetta] [reale]
632   true_negative  = confusion_matrix.iloc[0, 0] # caught_negatives
633   false_negative = confusion_matrix.iloc[1, 0] # missed_positives
634   false_positive = confusion_matrix.iloc[0, 1] # missed_negatives
635   true_positive  = confusion_matrix.iloc[1, 1] # caught_positives
636
637   # rapporto tra quelli che ho beccato e quelli che mi sono sbagliato a
    ↪  trovare
638   precision_pos = true_positive / (true_positive + false_positive)
639   precision_neg = true_negative / (true_negative + false_negative)
640
641   # rapporto tra quelli che ho beccato e quelli che avrei potuto trovare
642   recall_pos = true_positive / (true_positive + false_negative) #
    ↪  caught_pos / (caught_pos + missed_pos )
643   recall_neg = true_negative / (true_negative + false_positive) #
    ↪  caught_neg / (caught_neg + missed_neg )
644
645   f1_pos = (precision_pos * 2 * recall_pos) / (recall_pos + precision_pos)
646   f1_neg = (precision_neg * 2 * recall_neg) / (recall_neg + precision_neg)
647
648   if avg:
649     return {
650       "accuracy":  (true_positive + true_negative)/(true_positive +
        ↪  false_positive + true_negative + false_negative),
651       "precision": (precision_pos+precision_neg)/2,
652       "recall":    (recall_pos+recall_neg)/2,
```

```
653       "f1":        (f1_pos+f1_neg)/2
654     }
655   else:
656     return {
657       "accuracy":  (true_positive + true_negative)/(true_positive +
          ↪  true_negative + false_positive + false_negative),
658       "precision": (precision_pos,  precision_neg),
659       "recall":    (recall_pos,     recall_neg),
660       "f1":        (f1_pos,         f1_neg)
661     }
662
663 from statsmodels.stats.proportion import *
664
665 def calc_confidance_ranges(confusion_matrix:pd.DataFrame, confidence = 95):
666   """
667   calcola l intervallo di confidenza per 'accuracy', 'precision', 'recall'
668   https://www.statsmodels.org/dev/generated/statsmodels.stats.proportion.p
    ↪  roportion_confint.html
669   """
670   true_negative  = confusion_matrix.iloc[0, 0] # caught_negatives
671   false_negative = confusion_matrix.iloc[1, 0] # missed_positives
672   false_positive = confusion_matrix.iloc[0, 1] # missed_negatives
673   true_positive  = confusion_matrix.iloc[1, 1] # caught_positives
674
675   acc_range = proportion_confint(
676       true_positive + true_negative,
677       true_positive + true_negative + false_positive + false_negative,
678       alpha= 1 - confidence/100
679   )
680
681   # rapporto tra quelli che ho beccato e quelli che mi sono sbagliato a
      ↪  trovare
682   precision_pos = proportion_confint(true_positive,(true_positive +
      ↪  false_positive), alpha= 1 - confidence/100)
683   precision_neg = proportion_confint(true_negative,(true_negative +
      ↪  false_negative), alpha= 1 - confidence/100)
684
685   # rapporto tra quelli che ho beccato e quelli che avrei potuto trovare
686   recall_pos = proportion_confint(true_positive,(true_positive +
      ↪  false_negative) , alpha= 1 - confidence/100)
687   recall_neg = proportion_confint(true_negative,(true_negative +
      ↪  false_positive) , alpha= 1 - confidence/100)
688
689   return {
690     "accuracy":  acc_range,
```

```
691       "precision": (precision_pos,  precision_neg),
692       "recall":    (recall_pos,     recall_neg)
693     }
694
695   def get_sensitivity_and_specificity(confusion_matrix:pd.DataFrame):
696     cr = calc_confidance_ranges(confusion_matrix)
697     return {
698         "sensitivity": cr["recall"][0],
699         "specificity": cr["recall"][1]
700     }
701
702   def calc_p_values_for_matrix(confusion_matrix:pd.DataFrame, confidence =
      ↪  95):
703     """
704     calcola l intervallo di confidenza per 'accuracy', 'precision', 'recall'
705     https://www.statsmodels.org/dev/generated/statsmodels.stats.proportion.p
      ↪  roportion_confint.html
706     """
707     true_negative  = confusion_matrix.iloc[0, 0] # caught_negatives
708     false_negative = confusion_matrix.iloc[1, 0] # missed_positives
709     false_positive = confusion_matrix.iloc[0, 1] # missed_negatives
710     true_positive  = confusion_matrix.iloc[1, 1] # caught_positives
711
712     acc_range = proportions_ztest(
713         true_positive + true_negative,
714         true_positive + true_negative + false_positive + false_negative,
715         value= 1 - confidence/100
716     )
717
718     # rapporto tra quelli che ho beccato e quelli che mi sono sbagliato a
      ↪  trovare
719     precision_pos = proportions_ztest(true_positive,(true_positive +
      ↪  false_positive), value= 1 - confidence/100)
720     precision_neg = proportions_ztest(true_negative,(true_negative +
      ↪  false_negative), value= 1 - confidence/100)
721
722     # rapporto tra quelli che ho beccato e quelli che avrei potuto trovare
723     recall_pos = proportions_ztest(true_positive,(true_positive +
      ↪  false_negative) , value= 1 - confidence/100)
724     recall_neg = proportions_ztest(true_negative,(true_negative +
      ↪  false_positive) , value= 1 - confidence/100)
725
726     return {
727       "accuracy":  acc_range[1],
728       "precision": (precision_pos[1],  precision_neg[1]),
```

```
729     "recall":    (recall_pos[1],     recall_neg[1])
730   }
731
732 """## Funzioni per report delle metriche modelli
733
734 ### Calcolo matrice confidenza
735 """
736
737 def get_conf_matrices(model, data , what_to_evaluate = "test"):
738   """
739   dato un modello ed i dati su cui è addestrato
740   ritorna
741     - la matrice di confusione per le previsioni fatte con il modello
742     - la matrice di confusione per le previsioni fatte solo con la tac
743   """
744   data_index = 1 if what_to_evaluate == "test" else 0
745   Xs, ys = data
746   x = Xs[data_index]
747   y = ys[data_index]
748
749   model_cm = get_confusion_matrix(y, model.predict(x))
750   naive_cm = get_confusion_matrix(y, NaiveTACClassifier().predict(x))
751   return model_cm, naive_cm
752
753 """### Score Report"""
754
755 from scipy import stats
756
757 def confusion_matrices_accuracy_report(confusion_matrices_list, verbose =
    ↪  True):
758   """
759   calcola e stampa delle statistiche sulla accuratezza
760   """
761   accs_ranges = np.array([])
762   acc = np.array([])
763   for cm in confusion_matrices_list:
764     accs_ranges = np.append(accs_ranges,
      ↪  calc_confidance_ranges(cm)['accuracy'] )
765     acc = np.append(acc, get_scores_from_confusion_matrix(cm)['accuracy'] )
766
767   accs_ranges = accs_ranges.reshape((-1, 2))
768
769   avg_confidence = np.mean(accs_ranges, axis = 0)
770   avg_acc = np.mean(acc)
771   std_dev_acc = np.std(acc)
```

```
772   std_err_acc = stats.sem(acc)
773   report = f'Accuracy: {np.around(avg_acc,4)}±{np.around(std_err_acc,4)},
      ↪  std: {np.around(std_dev_acc,4)}, confidence:
      ↪  {np.around(avg_confidence,4)}'
774   if verbose: print(report)
775   return avg_acc, std_err_acc, std_dev_acc, avg_confidence
776
777 def confusion_matrices_sensitivity_report(confusion_matrices_list, verbose
    ↪  = True):
778   """
779   calcola e stampa delle statistiche sulla sensitivita ricavata da una
    ↪  lista di matrici di confusione
780   """
781   senss_ranges = np.array([])
782   sens = np.array([])
783   for cm in confusion_matrices_list:
784     senss_ranges = np.append(senss_ranges,
      ↪  calc_confidance_ranges(cm)['recall'][0] )
785     sens = np.append(sens,
      ↪  get_scores_from_confusion_matrix(cm)['recall'][0] )
786
787   senss_ranges = senss_ranges.reshape((-1, 2))
788
789   avg_confidence = np.mean(senss_ranges, axis = 0)
790   avg_sens = np.mean(sens)
791   std_dev_sens = np.std(sens)
792   std_err_sens = stats.sem(sens)
793   report = f'Sensitivity:
      ↪  {np.around(avg_sens,4)}±{np.around(std_err_sens,4)}, std:
      ↪  {np.around(std_dev_sens,4)}, confidence:
      ↪  {np.around(avg_confidence,4)}'
794   if verbose: print(report)
795   return avg_sens, std_err_sens, std_dev_sens, avg_confidence
796
797 def confusion_matrices_specificity_report(confusion_matrices_list, verbose
    ↪  = True):
798   """
799   calcola e stampa delle statistiche sulla specificità ricavata da delle
    ↪  matrici di confusione
800   (le matrici di confusione possono essere sia di un modello sia del
    ↪  classificatore 'naive')
801   """
802   specs_ranges = np.array([])
803   spec = np.array([])
804   for cm in confusion_matrices_list:
```

```
805       specs_ranges = np.append(specs_ranges,
          ↪  calc_confidance_ranges(cm)['recall'][1] )
806       spec = np.append(spec,
          ↪  get_scores_from_confusion_matrix(cm)['recall'][1] )
807
808     specs_ranges = specs_ranges.reshape((-1, 2))
809
810     avg_confidence = np.mean(specs_ranges, axis = 0)
811     avg_spec = np.mean(spec)
812     std_dev_spec = np.std(spec)
813     std_err_spec = stats.sem(spec)
814     report = f'Specificity:
        ↪  {np.around(avg_spec,4)}±{np.around(std_err_spec,4)}, std:
        ↪  {np.around(std_dev_spec,4)}, confidence:
        ↪  {np.around(avg_confidence,4)}'
815     if verbose: print(report)
816     return avg_spec, std_err_spec, std_dev_spec, avg_confidence
817
818 def confusion_matrices_negative_precision_report(confusion_matrices_list,
    ↪  verbose = True):
819     """
820     calcola e stampa delle statistiche sulla precisione ricavata da una
    ↪  lista di matrici di confusione
821     """
822     neg_prec_ranges = np.array([])
823     neg_prec = np.array([])
824     for cm in confusion_matrices_list:
825       if not np.isnan(get_scores_from_confusion_matrix(cm)['precision'][1]):
826         neg_prec_ranges = np.append(neg_prec_ranges,
            ↪  calc_confidance_ranges(cm)['precision'][1] )
827         neg_prec = np.append(neg_prec,
            ↪  get_scores_from_confusion_matrix(cm)['precision'][1] )
828
829     neg_prec_ranges = neg_prec_ranges.reshape((-1, 2))
830
831     avg_confidence = np.mean(neg_prec_ranges, axis = 0)
832     avg_neg_prec = np.mean(neg_prec)
833     std_dev_neg_prec = np.std(neg_prec)
834     std_err_neg_prec = stats.sem(neg_prec)
835     report = f'Positive Precision:
        ↪  {np.around(avg_neg_prec,4)}±{np.around(std_err_neg_prec,4)}, std:
        ↪  {np.around(std_dev_neg_prec,4)}, confidence:
        ↪  {np.around(avg_confidence,4)}'
836     if verbose: print(report)
837     return avg_neg_prec, std_err_neg_prec, std_dev_neg_prec, avg_confidence
```

```
838
839 def confusion_matrices_positive_precision_report(confusion_matrices_list,
    ↪ verbose = True):
840   """
841   calcola e stampa delle statistiche sulla sensitivita ricavata da una
    ↪ lista di matrici di confusione
842   """
843   pros_prec_ranges = np.array([])
844   pros_prec = np.array([])
845   for cm in confusion_matrices_list:
846     if not np.isnan(get_scores_from_confusion_matrix(cm)['precision'][0]):
847       pros_prec_ranges = np.append(pros_prec_ranges,
          ↪ calc_confidance_ranges(cm)['precision'][0] )
848       pros_prec = np.append(pros_prec,
          ↪ get_scores_from_confusion_matrix(cm)['precision'][0] )
849
850   pros_prec_ranges = pros_prec_ranges.reshape((-1, 2))
851
852   avg_confidence = np.mean(pros_prec_ranges, axis = 0)
853   avg_pros_prec = np.mean(pros_prec)
854   std_dev_pros_prec = np.std(pros_prec)
855   std_err_pros_prec = stats.sem(pros_prec)
856   report = f'Positive Precision:
    ↪ {np.around(avg_pros_prec,4)}±{np.around(std_err_pros_prec,4)}, std:
    ↪ {np.around(std_dev_pros_prec,4)}, confidence:
    ↪ {np.around(avg_confidence,4)}'
857   if verbose: print(report)
858   return avg_pros_prec, std_err_pros_prec, std_dev_pros_prec,
    ↪ avg_confidence
859
860 def print_confusion_matrices_report(confusion_matrices_list:list,
    ↪ calc_only_important = True) -> pd.DataFrame:
861   """
862   Stampa delle informazioni sulle performance di un modello
863   date alcune matrici di confidenza
864   """
865
866   if confusion_matrices_list == []: return None
867
868   statistical_values = ["average", "std. error", "std. deviation", "avg ci
    ↪ lower bound", "avg ci upper bound"]
869   dataframe_rows = []
870
871   avg, std_err, std_dev, (avg_ci_lower_b, avg_ci_upper_b) =
    ↪ confusion_matrices_accuracy_report(confusion_matrices_list)
```

```
872   dataframe_rows.append(pd.DataFrame(columns = statistical_values, index =
      ↪  ["accuracy"], data = [[avg, std_err, std_dev, avg_ci_lower_b,
      ↪  avg_ci_upper_b ]]))
873
874   if not calc_only_important:
875     avg, std_err, std_dev, (avg_ci_lower_b, avg_ci_upper_b)  =
        ↪  confusion_matrices_specificity_report(confusion_matrices_list)
876     spec = pd.DataFrame(columns = statistical_values, index =
        ↪  ["specificity"], data = [[avg, std_err, std_dev, avg_ci_lower_b,
        ↪  avg_ci_upper_b ]])
877
878   avg, std_err, std_dev, (avg_ci_lower_b, avg_ci_upper_b)  =
      ↪  confusion_matrices_sensitivity_report(confusion_matrices_list)
879   dataframe_rows.append(pd.DataFrame(columns = statistical_values, index =
      ↪  ["sensitivity"], data = [[avg, std_err, std_dev, avg_ci_lower_b,
      ↪  avg_ci_upper_b ]]))
880
881   avg, std_err, std_dev, (avg_ci_lower_b, avg_ci_upper_b) =
      ↪  confusion_matrices_positive_precision_report(confusion_matrices_list)
882   dataframe_rows.append(pd.DataFrame(columns = statistical_values, index =
      ↪  ["positive precision"], data = [[avg, std_err, std_dev,
      ↪  avg_ci_lower_b, avg_ci_upper_b ]]))
883
884   if not calc_only_important:
885     avg, std_err, std_dev, (avg_ci_lower_b, avg_ci_upper_b) = confusion_ma
        ↪  trices_negative_precision_report(confusion_matrices_list)
886     dataframe_rows.append(pd.DataFrame(columns = statistical_values, index
        ↪  = ["negative precision"], data = [[avg, std_err, std_dev,
        ↪  avg_ci_lower_b, avg_ci_upper_b ]]))
887
888   if not calc_only_important:
889     num_risparmiati = sum(list(map(lambda m:m[("predicted", "lett non
        ↪  grave")].sum(), confusion_matrices_list)))
890     num_operati =     sum(list(map(lambda m:m[("predicted", "lett
        ↪  grav")].sum(), confusion_matrices_list)))
891     print('Not Severe: ', num_risparmiati / (num_risparmiati +
        ↪  num_operati))
892
893   performance_dataframe = pd.concat(dataframe_rows)
894   if not calc_only_important:
895     r = (performance_dataframe.iloc[1,0] +
        ↪  performance_dataframe.iloc[2,0]) / 2
896     p = (performance_dataframe.iloc[3,0] +
        ↪  performance_dataframe.iloc[3,0]) / 2
897     print("F1-score:", 2 * (p*r)/(p+r))
```

```
898
899   return performance_dataframe
900
901 """## Addestramento dei modelli
902
903 Addestramento modelli con con Sklearn GridSearchCV
904 """
905
906 # creo un dizionario che
907 #  nelle chiavi conterrà il 'nome' di ogni modello
908 #  e nei valori avra delle liste
909 #   dentro ad ogni lista ci sono delle matrici di confusione relative al
    ↪  modello specificato nella chiave
910 #   ogni lista di matrici verrà usata per stampare delle statistiche sul
    ↪  modello
911 performance = {
912     model_name: []
913     for model_name in available_models
914 }
915 performance['naive'] = []
916
917 performance_train = {
918     model_name + '_train': []
919     for model_name in available_models
920 }
921
922 grids = {
923     model_name: []
924     for model_name in available_models
925 }
926
927
928 for i in range(5):
929   # prendo i dati
930   (X_train, X_test), (y_train, y_test) = get_split_data(test_size=1/3.5)
931   # stampo l'andamento
932   print(i, end = ' ')
933   # per ogni modello
934   for j, model_name in enumerate(available_models):
935     # addestro su i dati
936     # facendo un altro split dei dati all'interno di 'train_model' per
      ↪  validare il migliore
937     m, g = train_model(X_train, y_train, model_name, 0)
938     # valuto il modello su i dati mai visti
939     cm = get_confusion_matrix(
```

```
940             y_test,
941             np.sign( # uso la funzione segno per fare classificazione con
                ↪  regressori
942                 m.predict(X_test)-0.00010101 # certe volte la regressione
                    ↪  predice 0 e la f.segno non funziona
943                 )
944             )
945         # grids[model_name].append(cm)
946
947         performance[model_name].append(cm)
948         performance_train[model_name + '_train'].append(
949             get_confusion_matrix(
950               y_train,
951               np.sign( # uso la funzione segno per fare classificazione con
                  ↪  regressori
952                   m.predict(X_train)-0.00010101 # certe volte la regressione
                      ↪  predice 0 e la f.segno non funziona
953                   )
954               )
955         )
956
957     # calcolo per performance anche per il modello di _riferimento_
958     performance.get('naive').append(
959         get_confusion_matrix(
960             y_test,
961             NaiveTACClassifier().predict(X_test)
962         )
963     )
964
965 # dopo aver raccolto i risultati
966 # stampo delle statistiche sulle performance
967 for model_name, martices in performance.items():
968     print('---------')
969     print(model_name.upper())
970     print_confusion_matrices_report(
971         martices,
972         calc_only_important=False
973     )
974     print('(on train set)')
975     print_confusion_matrices_report(
976         performance_train.get(model_name + '_train', []),
977         calc_only_important=False
978     )
979
980 """Estraggo i modelli migliori per usarli poi nell'MLP"""
```

```
981
982 def get_best(array, func):
983   _l_sorting = lambda p: p[1] # seleziona il secondo di una tupla, che è
      ↪  il valore delle performance
984   _l_filter  = lambda x: x[0] is not 'naive'
985
986   # ritorna
987   return list(     # una lista
988       sorted(         # ordinata per '_l_sorting'
989         filter(_l_filter, # che esclude il modello di riferimento
990           map(              # dove ogni elemento è
991             lambda p:(        # una coppia fatta da
992                 p[0],            # nome modello,
993                 func(p[1], False)[0] # statistica modello
994                 ),
995             array
996           )
997         ),
998       key = _l_sorting, reverse = True
999     )
1000  )
1001
1002 # calcolo i migliori modelli per 'positive_precision'
1003 best_precision   = get_best(performance.items(),
      ↪  confusion_matrices_positive_precision_report)
1004 # calcolo i migliori modelli per 'positive_recall'/'sensitivity'
1005 best_sensitivity = get_best(performance.items(),
      ↪  confusion_matrices_sensitivity_report)
1006
1007 best_models = list(set( # rimuovo i possibili doppioni
1008     map(
1009         lambda x:x[0], # estraggo il nome
1010         best_sensitivity[:2] + best_precision[:3] # faccio un mix dei
          ↪  migliori modelli
1011     )
1012 ))
1013 best_models
1014
1015 """### MLP """
1016
1017 from keras.models import Model
1018 from keras.layers import Input, Dense, Conv1D, Dropout, Concatenate,
      ↪  Average, BatchNormalization
1019 from keras.optimizers import SGD
1020 from keras.callbacks import EarlyStopping, Callback
```

```
from keras.optimizers import Adam, SGD
import tensorflow as tf
from keras.metrics import Precision, Recall
from collections import deque

class GreatModelStop(Callback):
  """
  Una callback per smettere di addestrare se il modello è ritenuto sufficientemente accurato
  """
  def __init__(self, acc_to_stop_at):
    super(GreatModelStop, self).__init__()
    self.acc_to_stop_at = acc_to_stop_at

  def on_epoch_end(self, epoch, logs=None):
    current_val_accuracy = logs.get("val_accuracy")
    current_accuracy = logs.get("accuracy")
    val_acc_ok = current_val_accuracy >= self.acc_to_stop_at
    acc_ok = current_accuracy >= self.acc_to_stop_at + 0.04
    if val_acc_ok:
      self.model.stop_training = True

def enhance_dataset(X, pre_trained_models):
  """
  Data ogni istanza del dataset
  usa modelli pre-addestrati solo sui dati di train
  per creare un vettore composto dalle previsioni che i modelli avrebbero fatto su ogni stanza
  """
  return [
    [
     *[model.predict([xi])[0] for model in pre_trained_models],
     NaiveTACClassifier().predict([xi])[0]
    ]
    for xi in X
  ]

def calc_enahanced_dataset(best_models, test_size=1/3.5):
  # Divide il dataset normalmente
  (X_train, X_test), (y_train, y_test) = get_split_data(
      test_size = test_size,
      filtro_colonne=True
    )
  # addestra i migliori modelli su i dati di train
  pre_trained_models = [
```

```
1064            train_model(X_train, y_train, bm_name, 0)[0]
1065            for bm_name in best_models
1066      ]
1067
1068      # calcola un estensione del dataset con i modelli appena addestrati
1069      X_train_enh = enhance_dataset(X_train, pre_trained_models)
1070      X_test_enh = enhance_dataset(X_test, pre_trained_models)
1071
1072      X_train = (X_train, X_train_enh)
1073      X_test = (X_test, X_test_enh)
1074
1075      return (X_train, X_test), (y_train, y_test)
1076
1077  def keras_model_classification(data_input_size:int, model_input_size):
1078      """
1079      Crea un modello di classificazione con keras
1080      Notare che ha due input
1081          uno per le istanze del dataset
1082          uno per il vettore di previsioni creato con "enhance_dataset"
1083
1084      'keras_model_classification(14,4).summary()'
1085
1086      il modello ha due output, quindi deve essere addestrato su Y del tipo
      ↪  [1,0] / [0,1]
1087      """
1088      # strato di input
1089      data_input_layer = Input(shape=(data_input_size,))
1090      model_input_layer = Input(shape=(model_input_size,))
1091
1092      # secondo strato separato
1093      data_hidden_layer1 = Dense(int(data_input_size * 12 ), activation =
      ↪  'softmax')(data_input_layer)
1094      model_hidden_layer1 = Dense(int(model_input_size * 12 ), activation =
      ↪  'softmax')(model_input_layer)
1095
1096      # secondo strato separato
1097      data_hidden_layer2 = Dense(int(data_input_size * 10 ), activation =
      ↪  'softmax')(Dropout(0.15)(data_hidden_layer1)) # 10
1098      model_hidden_layer2 = Dense(int(model_input_size * 10 ), activation =
      ↪  'softmax')(Dropout(0.15)(model_hidden_layer1))
1099
1100      # terzo stato
1101      data_hidden_layer3 = Dense(30, activation =
      ↪  'elu')(Dropout(0.15)(data_hidden_layer2)) # BatchNormalization
```

```
1102    model_hidden_layer3 = Dense(20, activation =
        ↪  'elu')(Dropout(0.15)(model_hidden_layer2))# 30
1103
1104    concat_layer = Concatenate()([
1105                                  Dropout(0.1)(data_hidden_layer3),
1106                                  Dropout(0.1)(model_hidden_layer3),
1107    ])
1108
1109    last_hidden_layer1 = Dense(30, activation = 'relu')(concat_layer )
1110    last_hidden_layer2 = Dense(25, activation =
        ↪  'relu')(Dropout(0.1)(last_hidden_layer1) )
1111    # last_hidden_layer3 = Dense(6, activation =
        ↪  'relu')(Dropout(0.1)(last_hidden_layer2) )
1112
1113    output_layer = Dense(2, activation =
        ↪  'sigmoid')(Dropout(0.05)(last_hidden_layer2))
1114
1115    simple_model = Model(
1116        inputs=[data_input_layer, model_input_layer],
1117        outputs=output_layer,
1118        name="logreg_keras"
1119        )
1120    simple_model.compile(
1121        optimizer=Adam(learning_rate=0.00005),
1122        loss="binary_crossentropy",
1123        metrics=["accuracy"]
1124    )
1125    return simple_model
1126
1127
1128  # creo due variabili per memorizzare
1129  # le performance dei modelli addestrati
1130  matrices4 = []
1131  hists = []
1132
1133  # come sopra faccio piu test
1134  for i in range(10):
1135    print(i, end=')')
1136
1137    (X_train, X_test), (y_train, y_test) = calc_enahanced_dataset(
1138        best_models,
1139        test_size=1/3
1140      )
1141    X_train_1, X_train_2, X_test_1,  X_test_2 = np.array(X_train[0]),
        ↪  np.array(X_train[1]), np.array(X_test[0]), np.array(X_test[1])
```

```
1142    y_train, y_test = np.array(y_train), np.array(y_test)
1143
1144    # modifico le Y per metterle nella forma necessitata dal modello
1145    y_train_cat = np.array(list(map(lambda yi:[1,0] if yi==-1 else [0,1],
        ↪  y_train)))
1146    y_test_cat = np.array(list(map(lambda yi:[1,0] if yi==-1 else [0,1],
        ↪  y_test)))
1147
1148
1149    # istanzio il modello di keras
1150    final_model = keras_model_classification(
1151        X_train_1.shape[1],
1152        X_train_2.shape[1]
1153    )
1154
1155
1156    # faccio l'addestramento senza ulteriormente splittare i dati
1157    # e aggiungo l'andamento delle metriche del modello alla lista
        ↪  appropriata
1158    hist = final_model.fit(
1159        [ X_train_1, X_train_2 ], y_train_cat,
1160        epochs=300,
1161        validation_data=([ X_test_1, X_test_2 ], y_test_cat),
1162        verbose = 0,
1163        batch_size = 3,
1164        callbacks = [GreatModelStop(0.87)]
1165    )
1166    hists.append(hist)
1167    print(np.array(hist.history['val_accuracy'][-10:]).mean())
1168
1169    # calcolo la matrice di confusione
1170    matr = get_confusion_matrix(
1171        y_test,
1172        np.argmax( # mappa ogni previsione tra 0 e 1
1173            final_model.predict([ X_test_1, X_test_2 ]), axis = 1
1174        ) * 2 - 1 # poi mappa tra 0 e 2, poi tra -1 e 1, per comparare con
            ↪  le y originali
1175    )
1176    matrices4.append(matr)
1177
1178    # -- stampa le matrici di confusione relative al modello
1179    print(get_confusion_matrix(y_test,
        ↪  NaiveTACClassifier().predict(X_test_1)))
1180
1181 # stampa le statistiche sulle performance
```

```
_ = print_confusion_matrices_report(matrices4, False)

"""stampa altri dati sulle performance dei modelli addestrati"""

for m in matrices4: print(m)
[plt.plot(hist.history['val_accuracy']) for hist in hists]
[ np.array(hist.history['val_accuracy'][-10:]).mean() for hist in hists ]

"""### TabNet"""

from tabnet import TabNet, TabNetClassifier

def test_tabnet(output_dim = 5, feature_per_dec_step = 6, num_decision_steps = 2, epochs_100 = 8, n_exp = 2):
  hists2 = []
  matrices = []
  # come sopra faccio piu test
  for i in range(n_exp):
    # divido i dati
    (X_train, X_test), (y_train, y_test) = get_split_data(test_size = 1/3, trucchetto = False, filtro_colonne=False)

    # modifico le Y per metterle nella forma necessitata dal modello
    y_train_cat = np.array(list(map(lambda yi:[1,0] if yi==-1 else [0,1], y_train))).astype(np.float32)
    y_test_cat = np.array(list(map(lambda yi:[1,0] if yi==-1 else [0,1], y_test))).astype(np.float32)

    # converto il tipo dei dati
    X_train = X_train.astype(np.float32)
    X_test = X_test.astype(np.float32)

    tabnet_model = TabNetClassifier(
        feature_columns=None,
        num_classes = 2,
        num_features = X_train.shape[1],
        feature_dim = output_dim+feature_per_dec_step,
        output_dim = output_dim,
        num_decision_steps = num_decision_steps
    )
    tabnet_model.compile(optimizer=Adam(learning_rate = 0.0002), loss='binary_crossentropy', metrics=['accuracy'])

    # addestro il modello
    hist = tabnet_model.fit(
```

```
1222            X_train, y_train_cat, epochs=int(epochs_100*100),
1223            validation_data=(X_test, y_test_cat),
1224            verbose = 0
1225        )
1226        # e calcolo la matrice di confusione
1227        matr = get_confusion_matrix(
1228            y_test,
1229            np.argmax( # mappa ogni previsione tra 0 e 1
1230                tabnet_model.predict(X_test), axis = 1
1231            ) * 2 - 1 # poi mappa tra 0 e 2, poi tra -1 e 1, per comparare con
                ↪  le y originali
1232        )
1233        matrices.append(matr)
1234
1235        hists2.append(hist)
1236    return hists2, matrices
1237
1238 h,m = test_tabnet(output_dim = 6, feature_per_dec_step = 6, n_exp=5)
1239 [plt.plot(hist.history['val_accuracy']) for hist in h]
1240 print_confusion_matrices_report(m, False)
```