POLITECNICO DI TORINO

Corso di Laurea
in Matematica per l'Ingegneria

TESINA di MATEMATICA per l' INTELLIGENZA ARTIFICIALE

Introduzione alle Reti Neurali e analisi delle loro applicazioni a problemi di classificazione

Docente:
Francesco Vaccarino

Studente:
Marco Mungai Coppolino

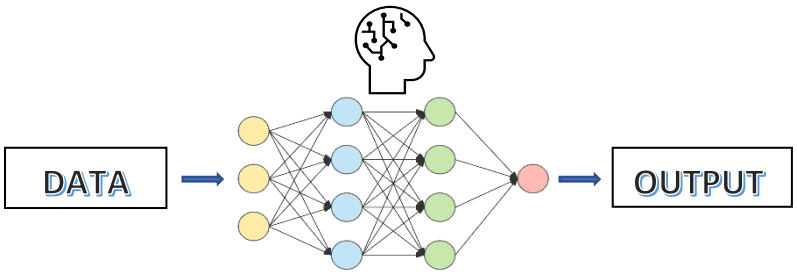
Anno Accademico 2021-2022

Indice

CAPITOLO 1: INTRODUZIONE ALLE RETI NEURALI

1.1 LE RETI NEURALI

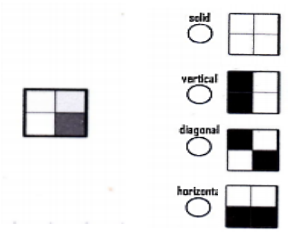
Le Reti Neurali (in inglese Neural Network) sono un modello computazionale matematico che costituiscono la base del Deep Learning, una sotto branchia del Machine Learning dove gli algoritmi sono ispirati dalla struttura del cervello umano.



Le reti neurali ricevono come input dei Data e si allenano per riconoscerne i pattern in essi racchiusi. In questo modo sono quindi capaci di predire gli output di altri insiemi di dati similari a quelli iniziali.

1.2 FUNZIONAMENTO RETE NEURALE IN BREVE

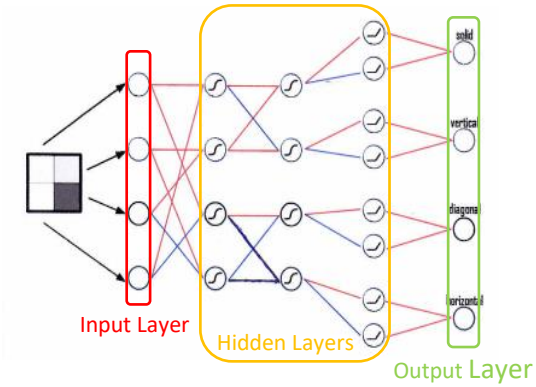
Per capire come funzionano le reti neurali si consideri l' esempio di una fotocamera capace di generare foto formate da 4 pixel, che possono essere solo o bianchi o neri o sfumature di grigio.



Si cerca quindi di trovare un modo per determinare in maniera automatica se una foto, scattata da questa fotocamera, corrisponda a un blocco solido (bianco o nero), una linea verticale, una linea diagonale o a una linea orizzontale.

Per riuscire nell' obiettivo si può ricorrere ai neural network.

Data infatti un' immagine, una rete neurale avrà una forma seguente.



I Neural Network sono costituiti da layer di neuroni, questi neuroni sono le unità di elaborazione della rete neurale

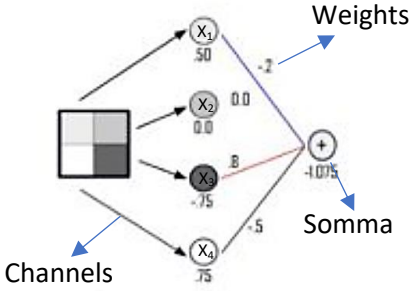
Inizialmente si ha l' Input Layer che riceve i dati dell' immagine in input.

L' Output Layer predice l' output finale della rete.

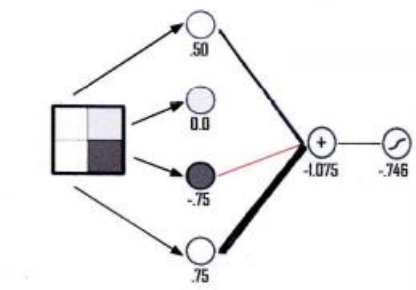
All' interno si trovano gli Hidden Layer che svolgono la maggior parte delle computazioni richieste dal nostro network.

Ciascun pixel della fotocamera è dato come input ad ogni neurone del primo layer, con valori che variano a seconda della sua luminosità, (In un range compreso tra [-1, 1]).

I neuroni di un layer sono collegati ai neuroni di un altro layer tramite dei **channel** (o **edge**), a ciascun channel è assegnato un valore numerico che prende il nome di **weight**.



Gli input sono moltiplicati per i corrispettivi pesi e la loro somma è inviata come input ai neuroni del primo hidden layer.



Ciascuno dei neuroni dell' hidden layer potrebbe essere già associato ad un altro valore numerico che prende il nome di **bias**, il quale viene aggiunto all' input somma.

$$(x_1 w_1 + x_2 w_2 + x_3 w_3 + x_4 w_4) + B_1$$

In ogni caso il totale viene poi passato ad una **threshold** function chiamata **funzione di attivazione**

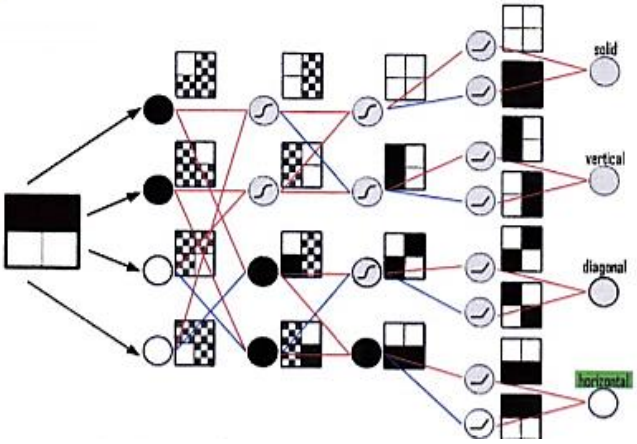
Si utilizza la funzione **sigmoide**, che ha il compito di comprimere i valori, in essa passati, nel range [-1, 1].

Si può semplificare il discorso considerando foto con pixel neri o bianchi e pesi uguali a -1 o 1. (In figura rappresenteremo in rosso i pesi **positivi** e in blu quelli **negativi**)

Ripetendo quindi ciò che si è fatto con l' input layer per i layer successivi, si costruisce in questo modo la rete neurale.

Una volta raggiunto l' ultimo hidden layer, invece che usare la sigmoide, si ricorre ad un' altra funzione di attivazione la **ReLU** (**Re**ctified **L**inear **U**nit).

Tale funzione mantiene gli stessi valori di input in essa passati, se positivi, e rende uguali a zero quelli negativi.



I data iniziali sono stati quindi propragati all' interno del network, in un processo che prende il nome di **Forward Propagation** (si tratta quindi di rete neurale **FEEDFORWARD**).
Nell' output layer il neurone con il più alto valore determina l' output del neural network.

1.3 LA FUNZIONE ERRORE E IL GRADIENTE

In generale si ha che i neural network non sono cosi precisi, spesso compiono delle previsioni errate. Ci si può quindi chiedere come si possa fare affidamento alle reti neurali vista la loro imprecisione. Il segreto è l' allenamento.

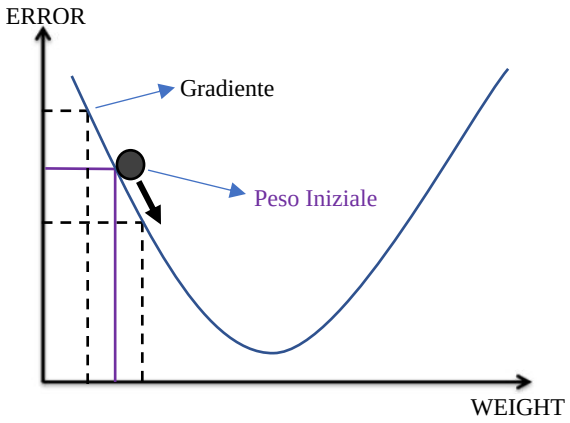
Il neural network va infatti incontro a un processo di training durante il quale, oltre all' input, gli forniamo l' output corretto da predire (la **Truth**)

Si chiama **ERROR** la differenza tra la truth e la previsione fatta dal neural network, la grandezza di tale errore indica di quanto la rete neurale sta sbagliando nelle sue previsioni.

L' idea è quella di aggiustare i vari weight (incrementandoli o diminuendoli) per fare in modo che l' errore sia il più basso possibile.

Per evitare di dover ricalcolare l' errore totale del neural network, ogni volta che si effettuano delle modifiche ai pesi, (processo questo molto dispendioso in termini computazionali), si procede con il determinare il **gradiente** della **curva errore**.

Si può cioè calcolare la direzione in cui si deve aggiustare i pesi senza dover tornare indietro e calcolare il nuovo errore.



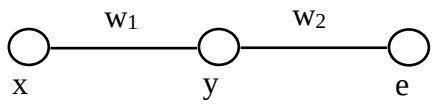
Il gradiente è dato dalla pendenza della retta tangente alla curva errore e si calcola nel seguente modo:

$$Slope = \frac{Variazione\ in\ Errore}{Variazione\ in\ Peso} = \frac{\partial(Error)}{\partial(Weight)}$$

Il problema è che per calcolare matematicamente il gradiente serve la funzione errore, ma non è sempre detto che la si conosca, si ricorre perciò ad un metodo alternativo per il calcolo del gradiente.

1.4 CHAINING E BACKPROPAGATION

Si consideri un semplice neural network formato da un singolo hidden layer.



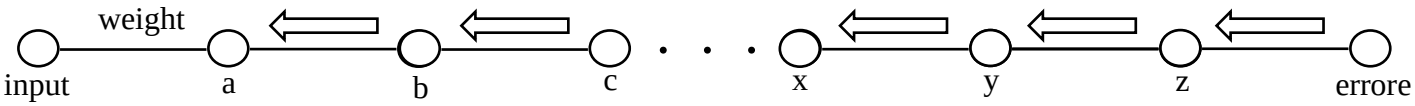
Dato l' input x si ha che: $y = x \cdot w_1 \Rightarrow \frac{\partial y}{\partial w_1} = x$

Inoltre: $e = y \cdot w_2 \Rightarrow \frac{\partial e}{\partial y} = w_2$

Si può quindi scrivere: $e = x \cdot w_1 \cdot w_2 \Rightarrow \frac{\partial e}{\partial w_1} = x \cdot w_2 = \frac{\partial y}{\partial w_1}\frac{\partial e}{\partial y}$

Questo dimostra che si può calcolare il gradiente di ogni piccolo step e moltiplicarli tutti insieme per ottenere il gradiente finale.

Tale processo prende il nome di **CHAINING** e per un neural network più complesso diventa qualcosa del genere:



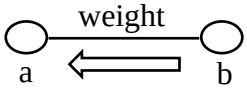
$$\frac{\partial errore}{\partial weight} = \frac{\partial a}{\partial weight} \cdot \frac{\partial b}{\partial a} \cdot \frac{\partial c}{\partial b} \cdot \frac{\partial d}{\partial c} \cdot \ldots \cdot \frac{\partial y}{\partial x} \cdot \frac{\partial z}{\partial y} \cdot \frac{\partial errore}{\partial z}$$

Per capire quindi quanto l' errore cambierà per una modifica di un peso del neural network si calcola la derivata di ogni piccolo step, procedendo all' indietro fino a raggiungere il peso interessato, e si moltiplicano fra loro le derivate.

Si compie quindi una **BACKPROPAGATION**, cioè si ripercorre il neural network procedendo da destra verso sinistra, riuscendo quindi ad evitare di dover calcolare la variazione totale dell' errore di tutto il sistema ogni qual volta si effettua una modifica.

Ci sono diverse operazioni di backpropagation da fare e per ciascuna di esse si deve calcolare il gradiente, il quale cambia a seconda del tipo di pezzetto di backpropagation che si sta calcolando.

- **Backpropagation WEIGHTS**: si consideri la weighted connection fra due neuroni a e b.

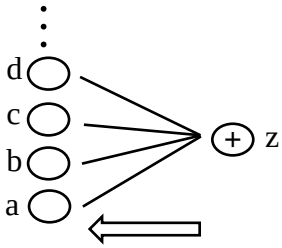


Si vuole calcolare $\frac{\partial errore}{\partial a}$ conoscendo $\frac{\partial errore}{\partial b}$

Si sfrutta il fatto che: $\frac{\partial errore}{\partial a} = \frac{\partial b}{\partial a} \cdot \frac{\partial errore}{\partial b}$

Di questa equazione non si conosce $\frac{\partial b}{\partial a}$, però si sa che $b = w \cdot a$, quindi derivando si ottiene $\frac{\partial b}{\partial a} = w$

- **Backpropagation SUMS**: un altro elemento della rete neurale sono i neuroni somma.

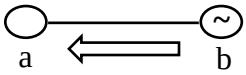


Si ha che: $\frac{\partial errore}{\partial a} = \frac{\partial z}{\partial a} \cdot \frac{\partial errore}{\partial z}$

Come per la Backpropagation WEIGHTS si conosce $\frac{\partial errore}{\partial z}$

Per calcolare $\frac{\partial z}{\partial a}$ si sfrutta il fatto che: $z = a + b + c + d + \cdots \Rightarrow \frac{\partial z}{\partial a} = 1$

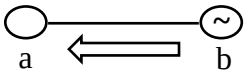
- **Backpropagation SIGMOID**: per la sigmoide si ha che: $\frac{\partial errore}{\partial a} = \frac{\partial b}{\partial a} \cdot \frac{\partial errore}{\partial b}$



La sigmoide b di a è data da: $\sigma(a) = \frac{1}{1+e^{-a}} = b$

Quindi $\frac{\partial b}{\partial a} = \sigma(a) \cdot (1 - \sigma(a))$

- **Backpropagation ReLU**: per la ReLU si ha infine che: $\frac{\partial errore}{\partial a} = \frac{\partial b}{\partial a} \cdot \frac{\partial errore}{\partial b}$



La ReLU b di a è data da: $\begin{cases} b = a\,, & a > 0 \\ b = 0\,, & a \leq 0 \end{cases}$

Quindi $\frac{\partial b}{\partial a} = \begin{cases} b = a\,, & a > 0 \\ b = 0\,, & a \leq 0 \end{cases}$

1.5 PROCESSO DI ALLENAMENTO

Conoscendo quindi tutti questi piccoli **backpropagation steps** e avendo l' abilità di unirli insieme si può calcolare l' effetto sull' errore del modificare un peso qualsiasi, questo per ogni vettore input fornito.

Per allenare il sistema si inizia con un neural network **completamente connesso**, senza conoscere cioè quale peso attribuire ai vari channel. Si procede assegnando quindi alle connessioni dei **pesi casuali**, creando un network arbitrario.

Sfruttando la truth si calcola l' errore del network e attraverso la backpropagation si procede all' indietro modificando i pesi nella giusta direzione (cioè quella che diminuisce l' errore).

Si ripete tale processo con degli **altri input** (migliaia o addirittura milioni se possibile) e eventualmente tutti i pesi raggiungeranno un punto in cui la rete neurale sarà capace di dare **risposte** molto simili alla **realtà**.

1.6 OBIETTIVO TESINA E METODI DI VALUTAZIONE PERFORMANCE

Le reti neurali artificiali permettono la modellizzazione di processi non lineari, cosa questa che le ha rese negli ultimi anni uno strumento molto popolare e utile per la risoluzione di vari problemi come la **classificazione**, **cluster** e **regression analysis**, **riconoscimento** di **pattern**, **riduzione** della **dimensione**, **rivelazione** di **anomalie** e molto altro.

Questo ampio spettro di abilità ha reso infatti possibile l' impiego dei neural network in molti campi, per esempio in medicina, programmi di riconoscimento facciale, bot per predizioni in borsa o anche per la produzione di composizioni musicali.

♦ **OBIETTIVO TESINA**: in questa tesina saranno testate le capacità di un particolare tipo di neural network il **Multi Layer Perceptron** (**MLP**), riguardo la risoluzione di un problema di classificazione.

I risultati della rete neurale verranno confrontati con quelli di un altro classificatore il **Support Vector Machine** (**SVM**).

Verrà infatti analizzato un dataset in cui i campioni sono divisi in varie classi di appartenenza (i **Target**), in base ai valori delle varie variabili da cui essi sono definiti (le **Features**) .

L' obiettivo è quello di allenare gli algoritmi sui campioni del dataset, di cui si conosce la classe di appartenenza, e poi testarli per vedere se sono in grado di prevedere la corretta classe di nuovi dati per cui la truth non è conosciuta.

♦ **DIVISIONE DATASET**: prima di usare gli algoritmi per fare previsioni su nuovi campioni è necessario testarne l' efficacia, per far ciò si separerà il dataset in tre sottoinsiemi:
Training set ($\mathcal{T}$), **Validation Set** ($\mathcal{V}$), **Test Set** ($\mathcal{P}$).

Dato un dataset formato da **coppie features-target** $\boldsymbol{D} = \{(\boldsymbol{x_1}, \boldsymbol{y_1}), \ldots, (\boldsymbol{x_i}, \boldsymbol{y_i})\} \subset \mathbb{R}^{\boldsymbol{D}} \boldsymbol{x}\, \mathbb{R}$, un **algoritmo di classificazione** e un insieme di **K modelli** $\widehat{\boldsymbol{f_1}}, \ldots, \widehat{\boldsymbol{f_K}}$ di tale classifier, caratterizati da diverse combinazioni di **iperparametri**, si avrà infatti la seguente divisione:

- **Training Set**: è l' insieme su cui si addestrano i vari modelli $\widehat{f_1}, \ldots, \widehat{f_K}$ dell' algoritmo.
- **Validation Set**: è l' insieme su cui si testano i K modelli e si identifica il migliore.
- **Test Set**: è l' insieme su cui si valutano le performance del modello migliore trovato, in modo da capire se possa essere usato per fare previsioni.

♦ **METODI DI VALUTAZIONE PERFORMANCE**: per riuscire a valutare e comparare le performance dei due classificatori si useranno una serie di parametri:

- **Accuracy**: l' accuratezza su un **Test Set** ($\mathcal{P}$) è il **rapporto** tra le **predizioni corrette** del classifier su $\mathcal{P}$ e il **numero** di **predizioni possibili**: Acc($\mathcal{P}$) = $\frac{num.\ pred.\ corrette\ su\ \boldsymbol{P}}{cardinalità\ di\ \boldsymbol{P}}$
  Non è sempre un indicatore esaustivo, specie nel caso di classi non bilanciate.

- **Matrice di Confusione A**: calcolata rispetto a $\mathcal{P}$ ha come elemento $\mathbf{a_{ij}}$ il **numero** di **campioni appartenenti** alla classe $\mathbf{C_i}$ **che** il **classifier** ha **predetto di appartenere alla classe** $\mathbf{C_j}$. Sulla diagonale principale della matrice si potranno vedere le predizioni corrette dell' algoritmo, mentre gli altri elementi indicheranno i numeri di errori fatti.

  Si useranno tre matrici di confusione, infatti, sia $\mathbf{a_{ij}}$ definito come sopra e sia **m** il **numero di classi** presenti nel dataset si ha allora che:

  - **Matrice di Confusione Non Normalizzata**: il valore della matrice di confusione non normalizzata, alle coordinate riga i, colonna j è dato da: $\mathbf{a_{ij}}$

  - **Matrice di Confusione Normalizzata rispetto le Vere Classi**: il valore della matrice di confusione normalizzata rispetto le vere classi, alle coordinate riga i, colonna j è dato da: $\frac{\mathbf{aij}}{(a_{i0}+a_{i1}+\dots+a_{ij}+\dots+a_{im})}$

  - **Matrice di Confusione Normalizzata rispetto le Classi Predette**: Il valore della matrice di confusione normalizzata rispetto le classi predette, alle coordinate riga i, colonna j è dato da: $\frac{\mathbf{aij}}{(a_{0j}+a_{1j}+\dots+a_{ij}+\dots+a_{mj})}$

- **Precision**: l'abilità del classificatore di non etichettare appartenente alla classe $\mathbf{C_i}$ un campione che non le appartiene:

  $$\mathbf{prec(C_i,\ \mathcal{P})} = \frac{num.\ di\ elem.\ correttamente\ predetti \in \mathbf{Ci}}{num.\ totale\ di\ elem.\ predetti \in \mathbf{Ci}} = \frac{\mathbf{a_{ii}}}{somma\ elem.\ in\ col.\ i-esima}$$

- **Recall**: l' abilità del classificatore di trovare tutti gli elementi appartenenti alla classe $\mathbf{C_i}$:

  $$\mathbf{rec(C_i,\ \mathcal{P})} = \frac{num.\ di\ elem.\ correttamente\ predetti \in \mathbf{Ci}}{num.\ totale\ di\ elem. \in \mathbf{Ci}\ in\ P} = \frac{\mathbf{a_{ii}}}{somma\ elem.\ in\ riga\ i-esima}$$

  Sia per la Precision che per la Recall si useranno i loro valori medi.

- **F-beta Score**: è la media pesata della Precision e la Recall, assume sempre valori compresi fra 0 e 1 (0 minimo, 1 massimo), e è un ulteriore parametro per misurare l' accuratezza del modello: $\boldsymbol{F_\beta(C_i, P)} = \mathbf{(1+\beta^2)}\frac{\mathbf{prec}(C_i,P)\cdot\mathbf{rec}(C_i,P)}{\boldsymbol{\beta^2}\mathbf{prec}(C_i,P)+\mathbf{rec}(C_i,P)}$
  Si utilizzerà l' $\mathbf{F_1}$ **Score**.

CAPITOLO 2: ANALISI DEL DATASET

2.1 IL DATASET WINE QUALITY

Il dataset analizzato è una raccolta di due dataset appartenenti alla Machine Learning Repository UCI, relativi a dei campioni di vino "vinho verde", provenienti dal nord del Portogallo.

I due database, uno riferito ai vini rossi e l' altro ai vini bianchi, contengono i dati necessari per lo studio della qualità di questi prodotti. Le variabili dei dataset, infatti, si dividono in 11 feature ottenute con dei test fisico-chimici e 1 feature basata su opinioni sensoriali di esperti nel settore che hanno diviso i vari campioni in una scala da 1 a 10, in ordine crescente di qualità.

A seguire un elenco delle variabili di input ciascuna accompagnata da una breve spiegazione:

TEST FISICO-CHIMICI (VARIABILI DI INPUT):

1. **fixed acidity**: gli acidi sono uno dei maggiori costituenti dei vini e contribuiscono notevolmente al loro sapore. Gli acidi si dividono in acidi fissi e volatili e risiedono maggiormente nei chicchi d' uva, ma si trasmettono anche al vino. Gli acidi fissi predominanti nei vini sono l' acido tartarico ($C_4H_6O_6$, solitamente compresi fra 1000 – 4000 mg/L), acido malico ($C_4H_6O_5$, 0 – 8000 mg/L), acido citrico ($C_6H_8O_7$, 0 – 500 mg/L), acido succinico ($C_4H_6O_4$, 500 – 2000 mg/L).
2. **volatile acidity**: generalmente legati al deterioramento del vino. Si tratta principalmente di acido acetico ($C_2H_4O_2$), le percentuali variano a seconda del vino, (per esempio per i vini rossi sono comprese fra 600 – 900 mg/L). Alti livelli possono comportare un sapore negativo tendente al vinegar.
3. **citric acid**: presente in piccole quantità, l' acido citrico ($C_6H_8O_7$) viene aggiunto ai vini per aumentarne l' acidità e donare freschezza.
4. **residual sugar**: sono dei residui rimasti dopo la fermentazione derivati da dei zuccheri naturali presenti nell' uva.
5. **chlorides**: sono dei sali minerali acidi presenti nei vini che contribuiscono a determinarne la sapidità
6. **free sulfur dioxide**: il biossido di zolfo SO_2, chiamato anche solforosa, viene utilizzato principalmente per le sue proprietà antiossidanti. Al ph del vino la solforosa si trova in equilibrio sotto diverse forme principalmente si distingue in SO_2 libera e SO_2 combinata. Solo una piccola parte della solforosa libera è attiva.
7. **total sulfur dioxide**: è la somma tra la quantità di SO_2 libera e combinata
8. **density**: la densità del vino (cioè il rapporto $\frac{Massa}{Volume}$) è molto simile a quello dell' acqua. Un litro di vino pesa circa 990 grammi.
9. **ph**: misura l' acidità/basicità del vino, in una scala che va da 0 a 14.
10. **sulphates**: additivi che contribuiscono ai livelli di SO_2.
11. **alcohol**: la percentuale di alcol contenuta nel vino.

TEST SENSORIALE (VARIABILE DI OUTPUT):

12. **quality**: i vini vengono classificati in una scala da 1 a 10 in ordine crescente in base alla loro qualità

2.2 IMPORTAZIONE E ANALISI PRELEMINARI

♦ **IMPORTAZIONE DATABASE**: a seguire si mostrano i due dataframe importati dalla repository UCI

red_wine database:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
1	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	5
2	7.8	0.880	0.00	2.6	0.098	25.0	67.0	0.99680	3.20	0.68	9.8	5
3	7.8	0.760	0.04	2.3	0.092	15.0	54.0	0.99700	3.26	0.65	9.8	5
4	11.2	0.280	0.56	1.9	0.075	17.0	60.0	0.99800	3.16	0.58	9.8	6
5	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	5
...	...	...	...	...	...	...	...	...	...	...	...	...
1595	6.2	0.600	0.08	2.0	0.090	32.0	44.0	0.99490	3.45	0.58	10.5	5
1596	5.9	0.550	0.10	2.2	0.062	39.0	51.0	0.99512	3.52	0.76	11.2	6
1597	6.3	0.510	0.13	2.3	0.076	29.0	40.0	0.99574	3.42	0.75	11.0	6
1598	5.9	0.645	0.12	2.0	0.075	32.0	44.0	0.99547	3.57	0.71	10.2	5
1599	6.0	0.310	0.47	3.6	0.067	18.0	42.0	0.99549	3.39	0.66	11.0	6

1599 rows × 12 columns

white_wine database:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
1	7.0	0.27	0.36	20.7	0.045	45.0	170.0	1.00100	3.00	0.45	8.8	6
2	6.3	0.30	0.34	1.6	0.049	14.0	132.0	0.99400	3.30	0.49	9.5	6
3	8.1	0.28	0.40	6.9	0.050	30.0	97.0	0.99510	3.26	0.44	10.1	6
4	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.99560	3.19	0.40	9.9	6
5	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.99560	3.19	0.40	9.9	6
...	...	...	...	...	...	...	...	...	...	...	...	...
4894	6.2	0.21	0.29	1.6	0.039	24.0	92.0	0.99114	3.27	0.50	11.2	6
4895	6.6	0.32	0.36	8.0	0.047	57.0	168.0	0.99490	3.15	0.46	9.6	5
4896	6.5	0.24	0.19	1.2	0.041	30.0	111.0	0.99254	2.99	0.46	9.4	6
4897	5.5	0.29	0.30	1.1	0.022	20.0	110.0	0.98869	3.34	0.38	12.8	7
4898	6.0	0.21	0.38	0.8	0.020	22.0	98.0	0.98941	3.26	0.32	11.8	6

4898 rows × 12 columns

♦ **CONTEGGIO CLASSI**: come specificato nella pagina web della repository UCI in cui è condiviso il dataframe (https://archive.ics.uci.edu/ml/datasets/wine+quality) al dataset wine-quality può essere associato un problema di classificazione.

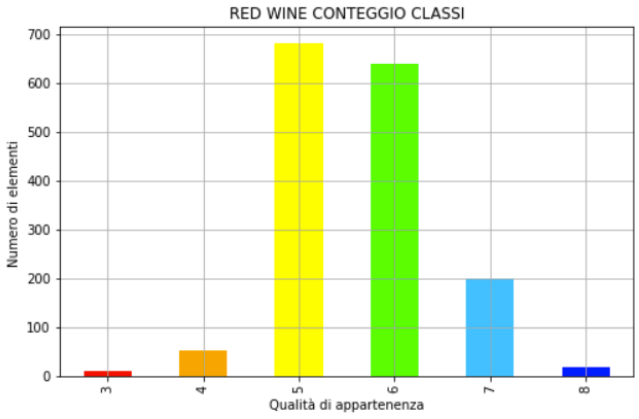
I vari vini, infatti, vengono divisi in classi tramite la feature "quality". L' obiettivo sarebbe quindi quello di testare vari algoritmi di classificazione in modo da creare un classifier abbastanza affidabile, capace di predire l' eccellenza di nuovi campioni di vino, attraverso i valori dei test fisico-chimici

Il problema non è però semplice, i due dataframe su cui allenare gli algoritmi **non** sono **bilanciati**, ci sono molti più vini aventi una qualità media rispetto a quelli con una cattiva o ottima qualità. Inoltre non si ha la certezza se tutte le **variabili** di **input** sono rilevanti, quindi sarebbe utile ricorrere a dei metodi di feature selection.

A seguire si riporta il **conteggio** del numero di campioni per ogni classe dei due dataframe, con annessa **frequenza** ($\frac{numero\ campioni\ di\ una\ classe}{numero\ totali\ di\ elementi\ del\ dataframe}$)

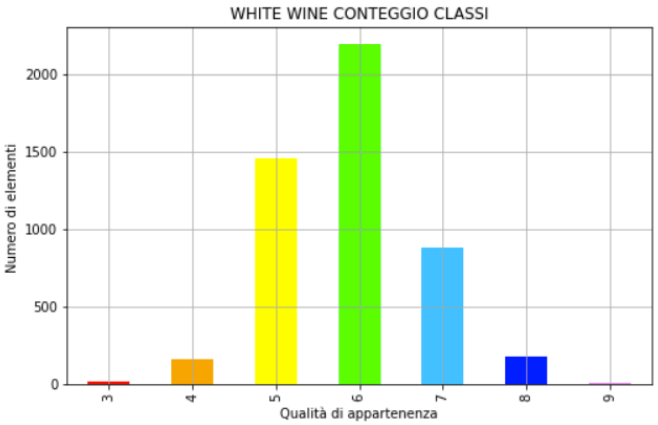
red_wine conteggio classi:

class	counts	freq.
3	10	0.006254
4	53	0.033146
5	681	0.425891
6	638	0.398999
7	199	0.124453
8	18	0.011257



white_wine conteggio classi:

class	counts	freq.
3	20	0.004083
4	163	0.033279
5	1457	0.297468
6	2198	0.448755
7	880	0.179665
8	175	0.035729
9	5	0.001021



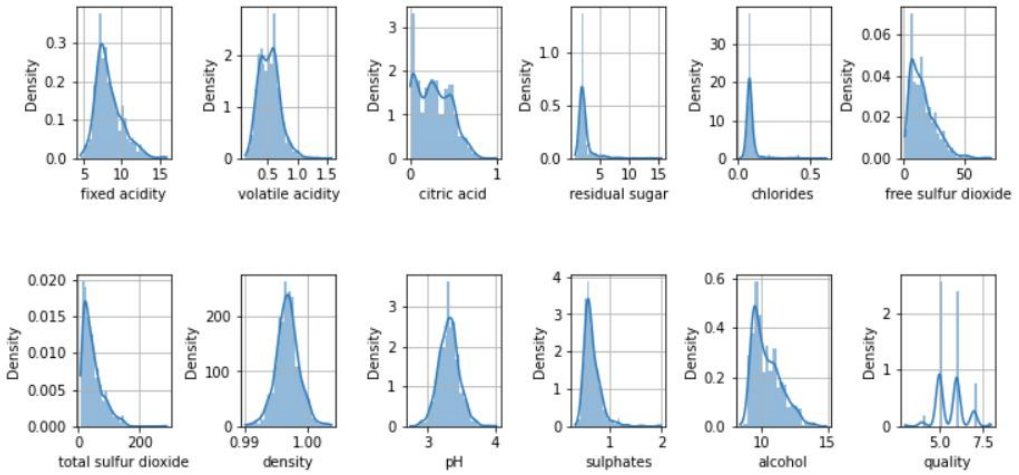
Come preannunciato c' è una grande discrepanza tra le classi più intermedie rispetto a quelle di valore più alto o basso. In particolare alcune classi non vengono nemmeno rappresentate da nessun campione (1, 2, 9, 10 per il DF red_wine e 1, 2, 10 per il DF white_wine)

♦ **STUDIO DELLA NORMALITÀ DEI DATAFRAME**: fra i vari strumenti di classificazione ci sono alcuni metodi come la LDA (Linear Discriminant Analysis) o la QDA (Quadratic Discriminant Analysis) che si basano sull' ipotesi che i dati dei dataframe seguano una distribuzione normale. Per verificare se i due database red_wine e white_wine rispettino o meno tale ipotesi si è ricorso a due metodi per lo studio della normalità.

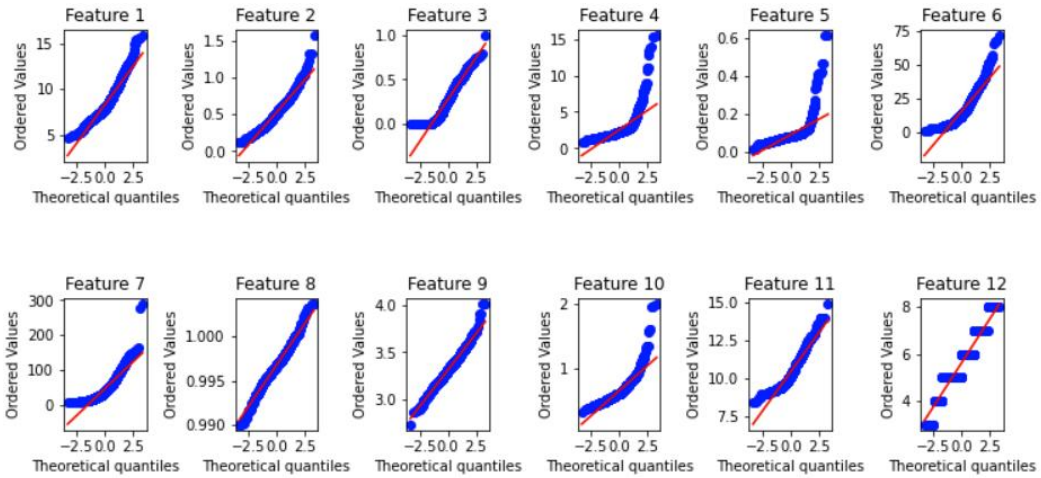
- **Histogram Plot**: i dati di ciascuna feature del DF vengono divisi in un numero specifico di intervalli e si procede con il rappresentare in un plot il numero di campioni con il loro intervallo di appartenenza.

- **Q-Q Plot**: il Q-Q Plot è la rappresentazione grafica dei quantili di una distribuzione. Viene cioè confrontata la distribuzione cumulata della variabile osservata con la distribuzione cumulata della normale. Se la feature osservata presenta una distribuzione normale, i punti di questa distribuzione congiunta si addensano sulla diagonale che va dal basso verso l'alto e da sinistra verso destra.

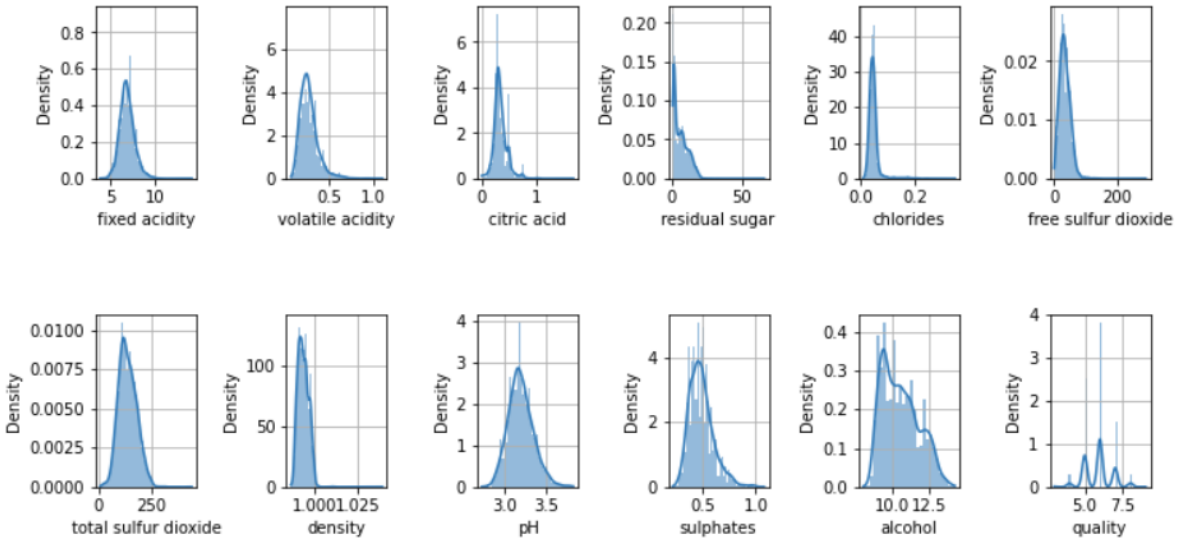
red_wine histogram plot:



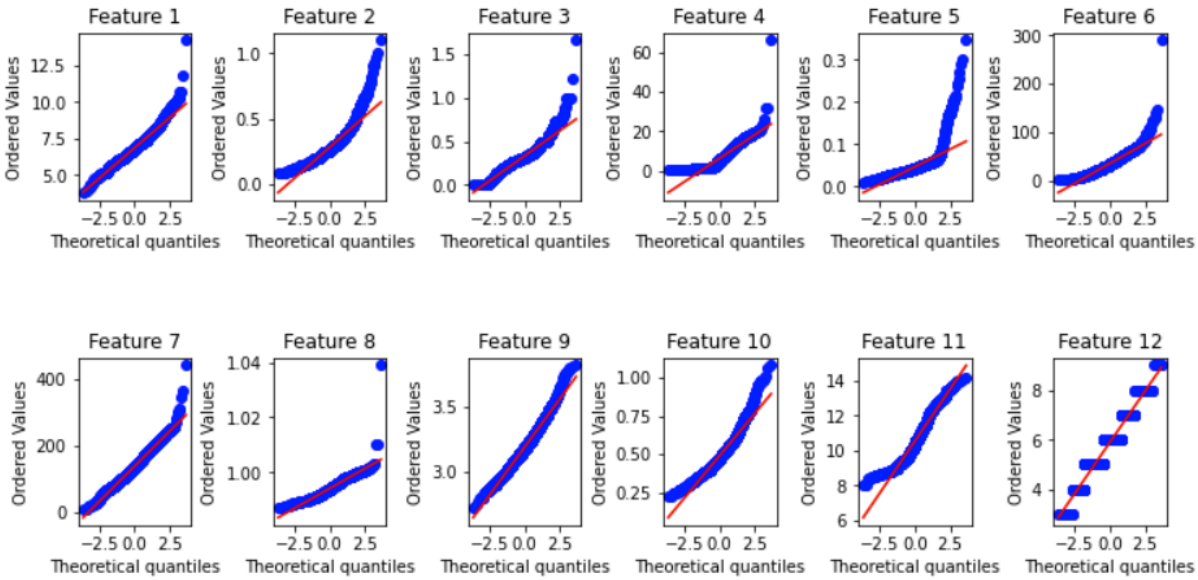
red_wine Q-Q plot:



white_wine histogram plot:



white_wine Q-Q plot:



Confrontando i vari grafici di entrambi i due metodi è evidente che alcune feature dei due dataframe si discostano notevolmente dalla distribuzione normale.

Guardando gli istogrammi si può infatti vedere che la maggior parte delle variabili di input sono caratterizzate dal fatto che alcuni campioni assumono dei valori molto diversi rispetto a quelli assunti dalla maggior parte degli elementi del dataframe.

Questo crea dei notevoli sbilanciamenti che sono poi rappresentati nei Q-Q plot tramite il discostamento fra le linee rosse della normale e le varie distribuzioni probabilistiche delle feature indicate in blu. Di conseguenza si può concludere che i due dataframe non rispettano l' ipotesi di normalità e quindi sarebbe sconsigliato ricorrere a classificatori come LDA o QDA.

2.3 RAPPRESENTAZIONE GRAFICA DEI DUE DATAFRAME

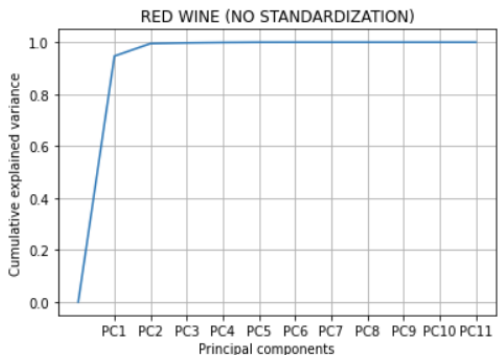
♦ PRINCIPAL COMPONENT ANALYSIS (PCA):

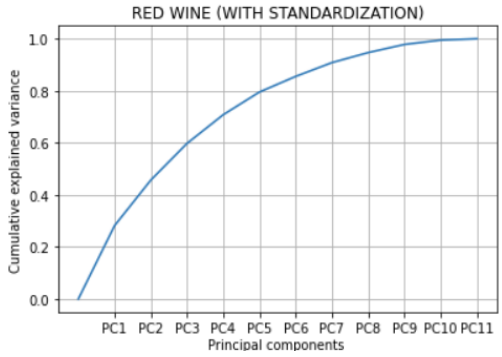
♣ **Applicazione PCA al dataset:** prima di poter applicare la PCA ai due dataset è importante osservare che c' è una differenza non indifferente tra gli ordini di grandezza della feature "total sulfur dioxide" e la feature "chlorides".

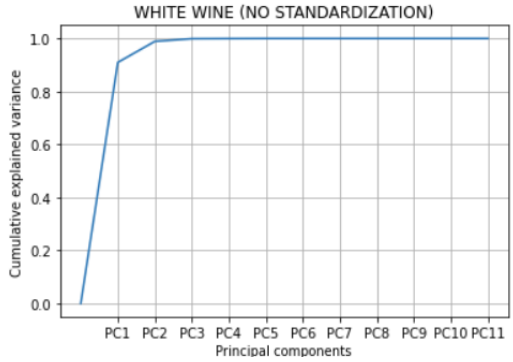
È quindi necessario effettuare una standardizzazione, altrimenti la maggior parte della varianza verrebbe "consumata" dalla feature "total sulfur dioxide".

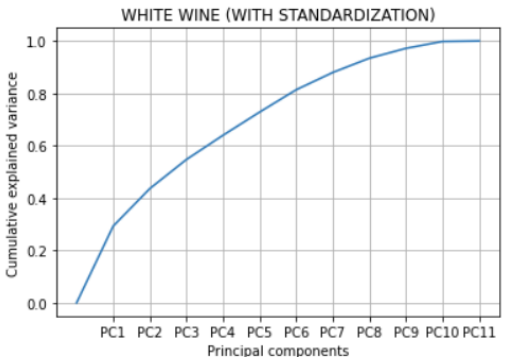
Si applica quindi lo StandardScaler() di sklearn che unisce la normalizzazione dei dati con il loro accentramento (sottraendo ai dati la media).

Si mostra quindi la percentuale di varianza spiegata dalle varie PC per i due dataset, sia nel caso con che senza standardizzazione.





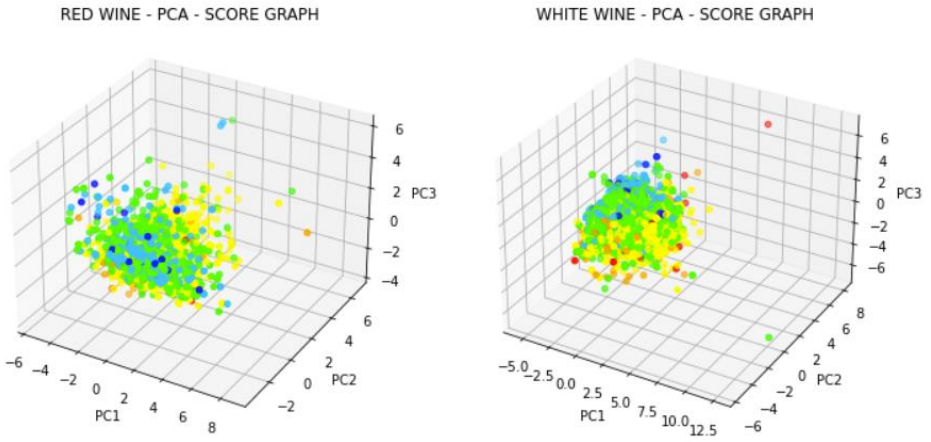




Si può vedere che nei casi in cui non è stata applicata la standardizzazione la maggior parte della varianza è stata "mangiata" dalle prime due componenti principali.

Per quanto riguarda invece il caso in cui viene applicata la standardizzazione è importante osservare che per ottenere una comprensibile rappresentazione del dataset tramite PCA è necessario usare un numero di componenti principali non indifferente.

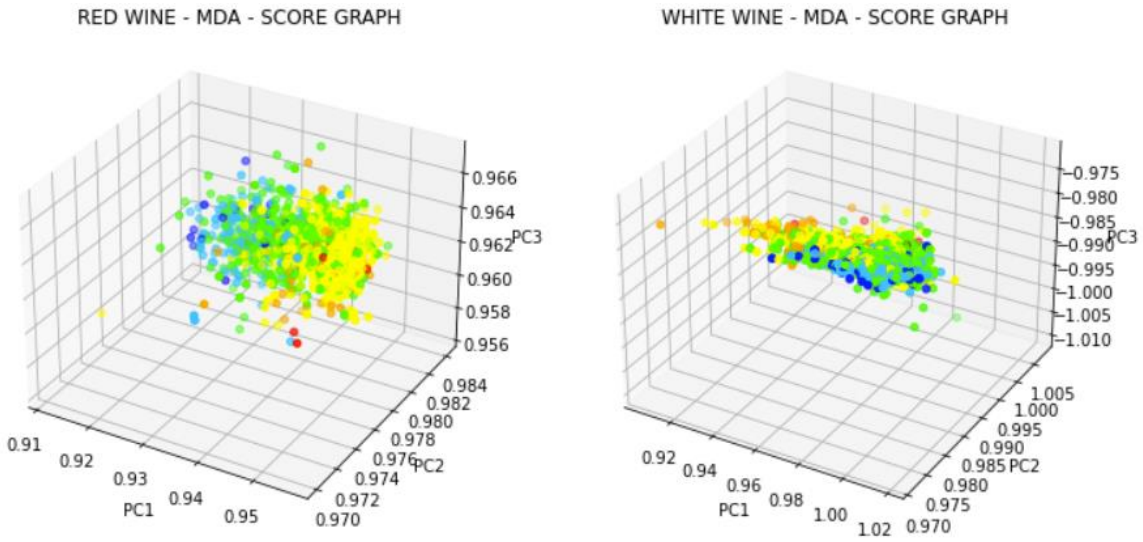
Si procede quindi a rappresentare direttamente lo **score graph** in $\mathbb{R}^3$, evitando il caso 2D



♦ **MULTIPLE DISCRIMINANT ANALYSIS (MDA):**

♣ **Applicazione MDA al dataset:** essendo i due database formati da molte classi (6 per il red_wine e 7 per il white_wine) possiamo tranquillamente proiettarlo in uno spazio a dimensione c-1 (dove c è il numero di classi del dataset)

Si procede quindi a rappresentare lo **score graph** della MDA in $\mathbb{R}^3$



Si può notare che il ricorrere alla MDA ha permesso di ottenere una separazione più netta delle varie classi.

2.4 PROBLEMA DI CLASSIFICAZIONE BINARIO

Il notevole non bilanciamento delle classi dei due dataset può rendere veramente problematica l' applicazione di metodi di classificazione, soprattutto considerando anche il poco numero di campioni a disposizione su cui allenare i due classifier.

Visto che l'obiettivo della tesina è quello di valutare le performance delle reti neurali, si procede allora con il dividere il principale problema di classificazione, in due diversi problemi:

- Il primo sarà un **problema di classificazione binario**, si andrà cioè ad unire le varie classi in due grandi classi, una contenente i vini con una qualità peggiore (quality da 1 a 5), l' altra con i vini con una qualità migliore (quality da 6 a 10).
- Il secondo sarà invece un **problema di multiclassificazione**, che verrà trattato nel paragrafo successivo.

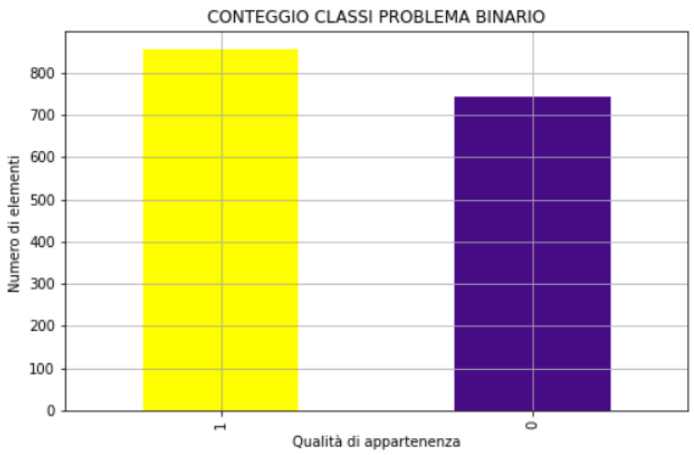
Questa divisione garantirà una analisi più completa e fornirà delle conclusioni più significative di quelle che invece si sarebbero ottenute se si fossero applicati i vari classifier direttamente ai due dataset.

Per il **problema binario** si analizzerà il database red_wine, lasciando il DF white_wine (con una dimensione maggiore rispetto al primo) al **problema multiclasse**.

PROBLEMA BINARIO: si procede quindi con il creare una copia del database red_wine e lo si divide in due classi nel seguente modo.

```
red_wine_copy = red_wine.copy()
red_wine_copy['quality'].replace([3, 4, 5], value=0, inplace=True)
red_wine_copy['quality'].replace([6, 7, 8], value=1, inplace=True)
```

Plottando il conteggio dei campioni delle due classi si può vedere come il problema binario sia abbastanza bilanciato.



SUPPORT VECTOR MACHINE: le SVM sono dei modelli di **supervised learning** associati ad algoritmi di apprendimento che analizzano dati, con l' obiettivo di classificarli.

Dato un **Training Set** $\mathcal{T} = \{(\mathbf{x_1}, y_1), (\mathbf{x_2}, y_2), \ldots, (\mathbf{x_N}, y_N)\} \subset \mathbb{R}^D \times \{\pm 1\}$ un **algoritmo** di **addestramento** per **SVM** è un esempio di modello $f: \mathbb{R}^D \to \{+1, -1\}$, che dopo un allenamento su $\mathcal{T}$ è capace di assegnare a nuovi esempi una delle due classi di appartenenza, effettuando quindi una **classificazione binaria**.

Sia $\pi_{w,b} \coloneqq \{\mathbf{x} \in \mathbb{R}^D | \mathbf{w^T x} + b = 0\}$ l' **iperpiano** di $\mathbb{R}^D$ definito dal **vettore normale** $\mathbf{w} \in \mathbb{R}^D$ e dal **parametro b**, ($\mathbf{w^T x}$ indica il prodotto scalare < **w**, **x** >).
Sia $dist(\pi_{w,b}, \mathbf{x}) = \frac{|\mathbf{w^T x + b}|}{||w||}$ la **distanza euclidea** tra un vettore $\mathbf{x} \in \mathbb{R}^D$ e l' iperpiano $\pi_{w,b}$.

L' **obiettivo** delle SVM è quello di trovare l' **iperpiano** $\pi_{w,b}$ che **meglio divide** i vettori $\mathbf{x_1}, \ldots, \mathbf{x_N}$ **nelle due classi di appartenenza ±1**, **massimizzando** cioè il **margine di separazione** $\boldsymbol{r} = dist(\pi_{w,b}, \mathbf{x_a})$, (dove $\mathbf{x_a}$ è il punto del dataset più vicino all' iperpiano).

Si vorrebbe quindi che i sample di $\mathcal{T}$ siano più lontani di **r** dall' iperpiano (nella rispettiva direzione a seconda che siano positivi o negativi).
Ciò si traduce con la seguente disuguaglianza: $y_i(\mathbf{w^T x_i} + \boldsymbol{b}) \geq \boldsymbol{r}$

Imponendo infine come ipotesi che il vettore **w** abbia norma unitaria, ($||\boldsymbol{w}|| = 1$), si ottiene un **problema** di **ottimizzazione vincolato**:

$$\begin{cases} \max_{w,b,r} r \\ \text{subject to } y_i(\mathbf{w^T x_i} + b) \geq r \,, \forall i = 1, \ldots, N \,, ||\mathbf{w}|| = 1 \,, r > 0 \end{cases}$$

▪ **Derivazione Tradizionale Del Margine**: senza perdere di generalità si può scegliere una **scala di misurazione** per i dati in modo che il valore del predittore $\mathbf{w^T x} + b$ sia uguale a 1 al sample più vicino all' iperpiano (cioè $\mathbf{x_a}$).

Sfruttando inoltre la bilinearità del prodotto scalare si dimostra che sotto queste ipotesi $r = \frac{1}{||\boldsymbol{w}||}$

Lasciando variare ||**w**|| ci si limita quindi a chiedere che i punti si trovino almeno a distanza 1 dall' iperpiano, invece che a distanza r, ovvero si usa la disequazione: $y_i(\mathbf{w^T x_i} + \boldsymbol{b}) \geq 1$

Considerando infine che massimizzare $r = \frac{1}{||\mathbf{w}||}$ è equivalente a minimizzare $\frac{1}{2}||w||^2 = \sum_{j=1}^{D} w_j^2$ il problema per trovare l' iperpiano separatore ottimale può essere così riscritto:

$$\textbf{HARD MARGIN SVM: } \begin{cases} \min_{w,b} \frac{1}{2}||\mathbf{w}||^2 \\ \text{subject to } y_i(\mathbf{w^T x_i} + b) \geq 1, \forall i = 1, \ldots, N \end{cases}$$

▪ **Il Problema Duale**: il problema Hard Margin SVM può essere risolto in maniera più efficiente passando alla sua formulazione duale.

Si introduce cioè una **variabile di slack $\boldsymbol{\alpha_i}$, $\forall$ vincolo**, che indica quanto il vincolo sia importante per la soluzione e si calcola la lagrangiana del problema:

$$\begin{cases} L(\mathbf{w}, \boldsymbol{\alpha}) = \frac{1}{2}\sum_{j=1}^{D} \mathrm{w_j}^2 - \sum_{i=1}^{N} \alpha_i (y_i \mathbf{w}^\mathbf{T} \mathbf{x_i} - 1) \\ s.t\ \alpha_i \geq 0, \forall i = 1, \ldots, N \end{cases}$$

Bisogna quindi **minimizzare** su **w** e **massimizzare su α** ottenendo:

$$\begin{cases} \max_{\boldsymbol{\alpha}} J(\boldsymbol{\alpha}) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \mathbf{x_i}^T \mathbf{x_j} \\ s.t\ \alpha_i \geq 0, \forall i = 1, \ldots, N, \ \sum_{i}^{N} \alpha_i\, y_i = 0 \end{cases}$$

Sia *SV* l' insieme degli indici dei **support vectors** (sono i sample più vicini all' iperpiano, possono essere più di uno) si ha quindi che la **funzione di decisione** è data da:

$$f(x) = \mathrm{sign}(\mathbf{w}^\mathbf{T}\mathbf{x}) = \mathrm{sign}\left(\sum_{\mathrm{i}\in \mathrm{SV}} \alpha_\mathrm{i} \alpha_\mathrm{j} \mathrm{y_i} \mathrm{y_j} \mathbf{x_i}^\mathbf{T} \mathbf{x_j} + \mathrm{b}\right), \text{dove b} = \frac{\mathbf{1}}{|\mathrm{SV}|} \sum_{\mathrm{i}\in \mathrm{SV}} (\mathrm{y_i} - \sum_{\mathrm{j}\in \mathrm{SV}} \alpha_\mathrm{j} \mathrm{y_j} \mathbf{x_i}^\mathbf{T} \mathbf{x_j})$$

Risolvendo il problema della lagrangiana si vanno a trovare gli α_i da inserire in J(**α**) e facendo così si hanno due casi possibili:

- Caso $\alpha_\mathrm{i} > 0$: risulta che $\mathrm{y_i}(\mathbf{w}^\mathbf{T}\mathbf{x_i}) = 1$ e quindi che il punto $\mathbf{x_i}$ è un **support vector**
- Caso $\alpha_\mathrm{i} = 0$: il punto $\mathbf{x_i}$ **non** è un **support vector**

Data la **soluzione ottimale $\boldsymbol{\alpha}^*$ del problema duale** si ha che i **pesi ottimali** sono:

$$\mathbf{w}^* = \sum_{\boldsymbol{i}\in \boldsymbol{SV}} \boldsymbol{\alpha}^*_{\boldsymbol{i}}\, \mathrm{y}_i \mathbf{x_i} \qquad b^* = \frac{\mathbf{1}}{|\mathrm{SV}|} \sum_{\mathrm{i}\in \mathrm{SV}} (\mathrm{y_i} - \mathbf{w}^{*\boldsymbol{T}} \boldsymbol{x_i})$$

▪ **Soft Margin SVM:** non sempre i campioni di un dataset sono linearmente separabili, è quindi necessario in questi casi introdurre una **tolleranza** sulla possibilità di oltrepassare il **margine** per i vettori $\mathbf{x_i}$.

Si introduce quindi una variabile di **slack** ξ_i, associata al campione ($\mathbf{x_i}$, $\mathrm{y_i}$), che permette al sample in questione di essere all' interno del margine o dal lato **sbagliato** dell' iperpiano.

Si procede quindi a **sottrarre** il valore di ξ_i dal **margine**, imponendo che $\xi_\mathrm{i} \geq 0$

Il problema può essere quindi riscritto nel seguente modo:

$$\textbf{SOFT MARGIN SVM: } \begin{cases} \min_{w} \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{N}\xi_i \\ \text{subject to } y_i(\mathbf{w^T x_i} + b) \geq 1 - \xi_i,\ \xi_i \geq 0, \forall i = 1, \ldots, N \end{cases}$$

Il parametro $C \in \mathbb{R}^+$ modifica la dimensione del margine e il quantitativo di slack totale (tanto più è **piccolo** è tanto più l' **SVM** sarà **morbida**).
C prende il nome di **parametro di regolarizzazione**, mentre $||\mathbf{w}||^2$ è il **regolarizzatore**.

▪ **Metodi Kernel**: se il dataset **non** è **linearmente separabile** si procede con il prendere i campioni del DF e li si **proietta** su uno **spazio a dimensione maggiore**.

Sia X l' insieme dei punti **x** del dataset, si costruisce infatti una **mappa** $\phi: X \subset \mathbb{R}^D \rightarrow \hat{X} \subset \mathbb{R}^M$ che **collega** l' **input space** con il **feature space** d' **arrivo**.
Una volta trovato il piano separatore tramite SVM, si procede con il farne la **controimmagine** tornando nella dimensione iniziale.

Se i punti del dataset sono difficili da separare, può succedere che la dimensione del feature space sia estremamente grande e non si può quindi usare $\phi(\mathbf{x_i})$ per calcolare la J(α) nella SVM.

In questi casi si trova quindi un **Kernel** K tale che $K(\mathbf{x_i}, \mathbf{x_j}) = <\phi(\mathbf{x_i}), \phi(\mathbf{x_j})>$, riuscendo quindi ad evitare di dover conoscere ϕ esplicitamente.

Ci sono vari tipi di kernel:

- **Kernel Lineare**: $K(\mathbf{x_i}, \mathbf{x_j}) = <\phi(\mathbf{x_i}), \phi(\mathbf{x_j})> = <\mathbf{x_i}, \mathbf{x_j}>$
- **Kernel Polinomiale**: $K(\mathbf{x_i}, \mathbf{x_j}) = <\phi(\mathbf{x_i}), \phi(\mathbf{x_j})> = <\mathbf{x_i}, \mathbf{x_j}>^d$
- **Kernel Gaussiano**: $K(\mathbf{x_i}, \mathbf{x_j}) = <\phi(\mathbf{x_i}), \phi(\mathbf{x_j})> = \exp\left(-\frac{1}{2\sigma^2}\left|\left|\mathbf{x_i} - \mathbf{x_j}\right|\right|_2^2\right)$
- **Kernel Sigmoidale**: $K(\mathbf{x_i}, \mathbf{x_j}) = <\phi(\mathbf{x_i}), \phi(\mathbf{x_j})> = \tanh(<\mathbf{x_i}, \mathbf{x_j}>)$

♦ **SVM E GRID SEARCH**:

♣ **Applicazione Grid Search**: prima di applicare la Grid Search si procede con il dividere il database nel seguente modo, tramite la funzione train_test_split di sklearn:

```
X = red_wine_copy.iloc[:, :-1].values
y = red_wine_copy['quality'].values
y_nomi_classe = ['classe 0', 'classe 1']

random_state = 20210526
test_p = 0.5
val_p = 0.4
indices = np.arange(X.shape[0])

ind_train, ind_test = train_test_split(indices, test_size=test_p, random_state=random_state, shuffle=True)
ind_train, ind_val = train_test_split(ind_train, test_size=val_p, random_state=random_state, shuffle=True)
```

Per avere una analisi approfondita si utilizzano 4 diversi kernel: **gaussiano** (**rbf**), **polinomiale** (**poly**), **sigmoidale** (**sigmoid**) e **lineare** (**linear**).

Inoltre, poiché la SVM minimizza il **vettore decisione w**, l' ottimale iperpiano che separa le classi è influenzato dalle dimensioni delle feature di input, ciò significa che è raccomandato che i **dati siano standardizzati** prima dell' allenamento del classifier.

Normalmente si farebbe il **fit** dello StandardScaler() sul training set e poi si procederebbe con l' usarlo per trasformare sia il training che il test.

Purtroppo applicando la GRID SEARCH la divisione del dataset in training e test set avviene internamente e si è infatti costretti a passarle in input il DF completo.

Per aggirare il problema si ricorre allo strumento **Pipeline** di sklearn. Le pipeline permettono di collegare in serie un numero arbitrario di elementi di processione dati, in modo tale che l' output di un elemento sia l' input di quello successivo.

Si può quindi creare una pipeline contenente prima lo StandardScaler() e poi il classifier SVC.

Passando tale pipeline alla GridSearchCV si riesce ad applicare lo scaling correttamente ai due set separati.

A seguire si riporta la GridSearchCV in modo da mostrare i range di parametri di input utilizzati:

```
n_features = X.shape[1]

C_list = [2 ** i for i in range(-2, 3)]
gamma_list = [1 / (i * n_features) for i in np.arange(0.5, 1.75, 0.5)]
ker_list = ['rbf', 'poly', 'sigmoid', 'linear']

pipe = Pipeline([('scaler', StandardScaler()), ('svc', SVC(class_weight='balanced'))])
hparameters = {'svc__kernel':ker_list, 'svc__C':C_list, 'svc__gamma':gamma_list}
pipe_parameters = {'pipe_kernel':ker_list}

svm_gs = GridSearchCV(pipe,
                      param_grid=hparameters,
                      scoring='f1_weighted',
                      return_train_score=True,
                      cv=zip([ind_train], [ind_val]),verbose=True)
svm_gs.fit(X, y)
```

```
Fitting 1 folds for each of 60 candidates, totalling 60 fits
GridSearchCV(cv=<zip object at 0x000002B01E2AD4C0>,
             estimator=Pipeline(steps=[('scaler', StandardScaler()),
                                       ('svc', SVC(class_weight='balanced'))]),
             param_grid={'svc__C': [0.25, 0.5, 1, 2, 4],
                         'svc__gamma': [0.18181818181818182,
                                        0.09090909090909091,
                                        0.06060606060606061],
                         'svc__kernel': ['rbf', 'poly', 'sigmoid', 'linear']},
             return_train_score=True, scoring='f1_weighted', verbose=True)
```

Si stampano quindi i risultati della GridSearchCV in modo da individuare quale kernel ha performato meglio e con quali parametri:

```
df_results = pd.DataFrame(svm_gs.cv_results_)

display(df_results.sort_values(['rank_test_score'], ascending=True))
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_svc__C	param_svc__gamma	param_svc__kernel	params	split0_test_score	mean_test_score	std_test_score	rank_test_score
40	0.002110	0.0	0.030652	0.0	2	0.090909	rbf	{'svc__C': 2, 'svc__gamma': 0.0909090909090909...	0.768497	0.768497	0.0	1
24	0.010088	0.0	0.030270	0.0	1	0.181818	rbf	{'svc__C': 1, 'svc__gamma': 0.1818181818181818...	0.762500	0.762500	0.0	2
56	0.010208	0.0	0.028539	0.0	4	0.060606	rbf	{'svc__C': 4, 'svc__gamma': 0.0606060606060606...	0.759185	0.759185	0.0	3
52	0.008172	0.0	0.022552	0.0	4	0.090909	rbf	{'svc__C': 4, 'svc__gamma': 0.0909090909090909...	0.759034	0.759034	0.0	4
16	0.010103	0.0	0.030282	0.0	0.5	0.090909	rbf	{'svc__C': 0.5, 'svc__gamma': 0.090909090909090...	0.753354	0.753354	0.0	5

Si osserva che la SVM che si è comportata meglio è stata quella con il kernel ‘rbf’ , il parametro C uguale a 2 e il parametro gamma pari a 0.090909.

Una volta individuata l’ SVM migliore si continua con il riaddestrarla sul training set del dataframe, per poi testarla sul validation e il test set. A seguire i risultati:

	Accuracy	Precision	Recall	F1
training	0.835073	0.841927	0.835073	0.834984
validation	0.768750	0.780129	0.768750	0.768497
test	0.735000	0.747999	0.735000	0.734418

Matrice di Confusione No Normalizzazione:

	classe 0	classe 1
classe 0	303	67
classe 1	145	285

Matrice di Confusione Normalizzata rispetto le Vere Classi:

	classe 0	classe 1
classe 0	0.818919	0.181081
classe 1	0.337209	0.662791

Matrice di Confusione Normalizzata rispetto le Classi Predette:

	classe 0	classe 1
classe 0	0.676339	0.190341
classe 1	0.323661	0.809659

MULTI LAYER PERCEPTRON: gli **MLP** sono un tipo di rete neurali **feedforward** caratterizzato da una sequenza di **layer completamente connessi**.

▪ **Fully-Connected Layer**: si prenda per esempio un **layer** L formato da **m** nodi, e una funzione di attivazione σ: $\mathbb{R} \to \mathbb{R}$ (per esempio la sigmoide). Si avrà che il layer L sarà completamente connesso al precedente layer di **n** unità tramite la **funzione caratterizzante** $\mathcal{L} : \mathbb{R}^m \to \mathbb{R}^n$ così definita: $\mathcal{L}(\mathbf{x}) \coloneqq \sigma(\mathrm{W}(\mathbf{x}) + \mathbf{b})$, $\forall x \in \mathbb{R}^n$ dove si ha che:

- $\mathbf{W} \in \mathbb{R}^{nxm}$ è la **matrice dei pesi**
- $\mathbf{b} \in \mathbb{R}^m$ è il **vettore bias**

▪ **Funzione Parametrica**: dato un MLP, formato da un **input layer** $L^{(0)}$ di **n** unità, $L^{(1)}$,…, $L^{(H)}$ **hidden layer** e da un **output layer** $L^{(H+1)}$ di **m** unità, si può rappresentare la rete neurale tramite la **funzione caratterizzante**: $\hat{\mathrm{F}}(\mathbf{x})$: $\mathbb{R}^n \xrightarrow{\mathcal{L}(1)} \mathbb{R}^{n_1} \xrightarrow{\mathcal{L}(2)} \ldots \xrightarrow{\mathcal{L}(\mathrm{H})} \mathbb{R}^{n_H} \xrightarrow{\mathcal{L}(\mathrm{H}+1)} \mathbb{R}^m$

▪ **Metodo Steepest Descent**: i metodi di discesa del gradiente si pongono come obiettivo la minimizzazione di **funzioni di perdita** f: $\mathbb{R}^n \to \mathbb{R}$
Sfruttando il fatto che $-\nabla f(\mathbf{w_0})$ è la direzione di **più rapida discesa** della funzione f nel punto $\mathbf{w_0}$, tali metodi permettono di trovare la soluzione del problema: $\min_{\mathbf{x} \in \mathbb{R}^n} f(\mathbf{w})$
Lo **Steepest Descent** ne è un esempio, dato un punto di partenza $\mathbf{x_0} \in \mathbb{R}^n$ si avrà che il **passo i-esimo** dell' algoritmo è dato da:

$$\mathbf{w_{i+1}} = \mathbf{w_i} - \alpha_i \nabla f(\mathbf{w_i}) \text{ , } \forall \mathrm{i} \geq 0 \text{ dove } \alpha_i \in \mathbb{R}^+ \text{ è un } \textbf{fattore di moltiplicazione}$$

▪ **Funzione Loss**: data quindi una **funzione target** $\mathrm{y} = \mathrm{F}(\mathbf{x})$ per addestrare un MLP con una funzione parametrica $\widehat{\mathrm{F}_{\mathrm{w}}}(\mathbf{x})$ si vuole minimizzare la **funzione loss:**

$$f(\mathbf{w}) = \mathrm{Loss}(\mathbf{w}) \coloneqq \sum_{\mathbf{x} \in \mathbb{R}^n} |\mathrm{F}(\mathbf{x}) - \widehat{\mathrm{F}_{\mathrm{w}}}(\mathbf{x})|$$

Si applica quindi il **metodo Steepest Descent,** (dopo aver usato la backpropagation per calcolare i vari gradienti), per **trovare i pesi che minimizzano la loss**.

▪ **Metodo Mini-Batch**: quando si applica tale metodo si effettua una **K-partizione casuale** sul **training set** $\mathcal{T}$. Si procede cioè a dividere $\mathcal{T}$ in delle **minibatch** B_1, …, B_k .
Si inizia quindi l' addestramento su B_1 per poi usarne l' esito per addestrare B_2 e così via.
Le minibatch permettono di effettuare molti addestramenti su pochi punti, invece che singoli addestramenti su miliardi di punti, e questo è perfetto nel caso del dataset wine_quality formato da pochi campioni.

♦ **MULTI LAYER PERCEPTRON (MLP)** :

♣ **Applicazione Multi Layer Perceptron:** come fatto per la SVM prima di applicare il classifier si esegue una separazione del DF in training, validation e test set:

```
random_state = 20210526
np.random.seed(random_state)

test_p = 0.5
val_p = 0.2

X_trainval, X_test, y_trainval, y_test = train_test_split(X, y, test_size=test_p, random_state=random_state, shuffle=True)

display(pd.DataFrame({'X_trainval': X_trainval.shape, 'X_test': X_test.shape}, index=['N. sanmples', 'N.features']))
```

	X_trainval	X_test
N. sanmples	799	800
N.features	11	11

Anche per le reti neurali è importante effettuare uno scaling dei dati, per fortuna diversamente da quanto fatto per la GridSearchCV non è necessario utilizzare una pipeline

```
standard_scaler = StandardScaler()
X_trainval = standard_scaler.fit_transform(X_trainval)
X_test = standard_scaler.transform(X_test)
```

Si procede costruendo un MLP formato da 5 hidden layers, ciascuno con 256 unità e funzione di attivazione **ReLU**. I parametri di addestramento sono i seguenti:

1) **Batch Size**: 4

2) **Massimo Numero Di Epoche**: 5000

3) **Early Stopping**: 75 epoche di pazienza

Si riporta l' addestramento del neural network:

```
hidden_layer_sizes = [256] * 5
activation = 'relu'
patience = 75
max_epochs = 5000
verbose = False
batch_sz = 4

mlp = MLPClassifier(hidden_layer_sizes=hidden_layer_sizes,
                    activation=activation,
                    early_stopping=True,
                    n_iter_no_change=patience,
                    max_iter=max_epochs,
                    validation_fraction=val_p,
                    batch_size=batch_sz,
                    verbose=verbose,
                    random_state=random_state,
                    solver='adam',
                    # learning_rate='adaptive'
                   )
```

```
mlp.fit(X_trainval, y_trainval)
```

```
MLPClassifier(batch_size=4, early_stopping=True,
              hidden_layer_sizes=[256, 256, 256, 256, 256], max_iter=5000,
              n_iter_no_change=75, random_state=20210526,
              validation_fraction=0.2)
```

Dopo aver addestrato il neural network si valutano le sue performance e se ne calcola l' accuracy sui vari set:

	Accuracy	Precision	Recall	F1
train. + val.	0.906133	0.908098	0.906133	0.906228
test	0.770000	0.770454	0.770000	0.770150

Matrice di Confusione No Normalizzazione:

	classe 0	classe 1
classe 0	282	88
classe 1	96	334

Matrice di Confusione Normalizzata rispetto le Vere Classi:

	classe 0	classe 1
classe 0	0.762162	0.237838
classe 1	0.223256	0.776744

Matrice di Confusione Normalizzata rispetto le Classi Predette:

	classe 0	classe 1
classe 0	0.746032	0.208531
classe 1	0.253968	0.791469

♦ **CONCLUSIONI** : sia la GridSearchCV che il MLP hanno ottenuto dei discreti risultati nella classificazione del problema binario.

La rete neurale in questo caso è riuscita a outperformare leggermente la support vector machine, ottenendo una accuracy pari a 0.77 rispetto alla suo controparte che si è limitata a un risultato pari a 0.735.

2.5 PROBLEMA DI MULTICLASSIFICAZIONE

In questo paragrafo si ripetono i passaggi fatti nel problema binario applicandoli al dataset white_wine e al problema multiclasse.

Come fatto in precedenza andremo ad unire le classi, dividendole però questa volta in 3 gruppi, trattando quindi un problema più simile a quello originale, ma riuscendo comunque a ottenere dei risultati significativi.

I vini di qualità 3, 4 e 5 formeranno la classe 0, quelli di qualità 6 la classe 1 e quelli di qualità 7, 8, 9 la classe 2

Come fatto nel problema binario si crea una copia del dataset white_wine e si effettuano le modifiche sopra citate:

```
white_wine_copy = white_wine.copy()
white_wine_copy['quality'].replace([3, 4, 5], value=0, inplace=True)
white_wine_copy['quality'].replace([6], value=1, inplace=True)
white_wine_copy['quality'].replace([7, 8, 9], value=2, inplace=True)
```

Plottando il conteggio dei campioni delle due classi si può vedere come il problema multiclasse rimanga comunque abbastanza non bilanciato nonostante l' aver accorpato le classi con meno elementi:

♦ **SVM E GRID SEARCH**: nella sua forma più semplice l' SVM viene applicata a problemi di classificazione binaria, dove i dati vengono divisi in due classi.

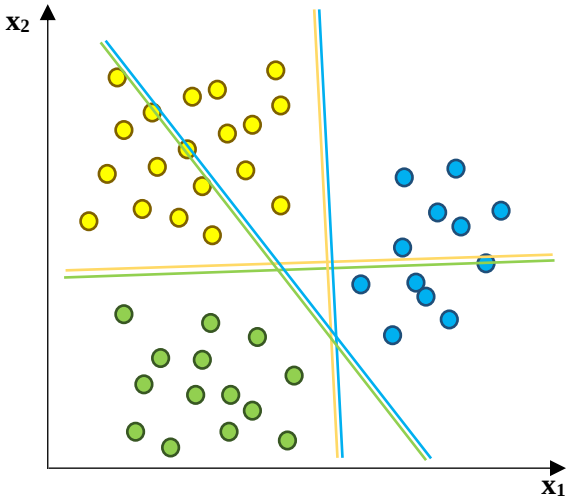
In caso di un numero più alto di classi si ricorre allo stesso principio, il problema multiclasse viene infatti diviso in più problemi binari.

Ci sono due metodi diversi per trattare tale divisione:

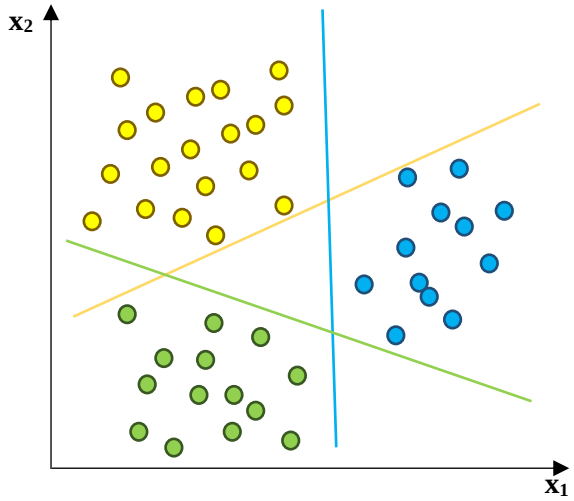
1) **Metodo One-VS-One (OvO)**: in questo metodo viene associata una SVM binaria ad ogni coppia di classi. Ogni SVM va quindi a generare un iperpiano che separa la sua coppia di classi, ignorando i punti della terza classe.

   Sia **m** il numero di classi, in tale approccio il multiclassifier andrà quindi ad addestrare $\binom{m}{2}$ SVM binarie per ciascuna coppia per un totale di $\frac{m(m-1)}{2}$ SVM.

   Immaginando che i punti del dataset abbiano solo 2 dimensioni invece che 11 tale classificazione darebbe qualcosa del genere:



2) **Metodo One-VS-Rest (OvR)**: in questo metodo si vanno ad utilizzare tante SVM binarie quante sono le classi. Viene infatti generato per ogni classe un iperpiano che la separa dai punti di tutte le altre classi.



♣ **Alcuni Accorgimenti**: per contrastare l' imbilanciamento del dataframe si ricorrerà ad un range di parametri C e gamma più ampio rispetto a quello usato nel problema binario.

Un altro accorgimento che dovrebbe migliorare leggermente i risultati sarebbe quello di sfruttare l' attributo class_weight='balanced' del classifier SVC di Sklearn. Tale attributo modifica il parametro C assegnando invece alla classe i il parametro $class\_weight[i] \cdot C$ La modalità 'balanced' modifica il peso attribuito ad ogni classe in modo inversamente proporzionale alla frequenza di tale classe nei campioni del DF.

Come nel caso binario si ricorre inoltre alla pipeline per effettuare lo scaling dei dati.

A seguire la creazione del training, validation e test set (si usa lo stesso split per i due metodi per comparare i due risultati):

```
X = white_wine_copy.iloc[:, :-1].values
y = white_wine_copy['quality'].values

y_nomi_classe = ['classe 0', 'classe 1', 'classe 2']
random_state = 20210526
test_p = 0.5
val_p = 0.4
indices = np.arange(X.shape[0])

ind_train, ind_test = train_test_split(indices, test_size=test_p, random_state=random_state, shuffle=True)
ind_train, ind_val = train_test_split(ind_train, test_size=val_p, random_state=random_state, shuffle=True)
```

♣ **One-VS-One Classifier**: il classifier SVC di scikit-learn implementa in modo automatico l' approccio One-VS-One.

```
n_features = X.shape[1]

C_list = [2 ** i for i in range(-2, 4)]

gamma_list = [1 / (i * n_features) for i in np.arange(0.15, 1.75, 0.5)]
ker_list = ['rbf', 'poly', 'sigmoid', 'linear']

pipe = Pipeline([('scaler', StandardScaler()), ('svc', SVC(class_weight='balanced'))])
hparameters = {'svc__kernel':ker_list, 'svc__C':C_list, 'svc__gamma':gamma_list}

svm_gs = GridSearchCV(pipe,
                      param_grid=hparameters,
                      scoring='f1_weighted',
                      return_train_score=True,
                      cv=zip([ind_train], [ind_val]),
                      verbose=True)
svm_gs.fit(X, y)
```

```
Fitting 1 folds for each of 96 candidates, totalling 96 fits
GridSearchCV(cv=<zip object at 0x0000026D88823240>,
             estimator=Pipeline(steps=[('scaler', StandardScaler()),
                                       ('svc', SVC(class_weight='balanced'))]),
             param_grid={'svc__C': [0.25, 0.5, 1, 2, 4, 8],
                         'svc__gamma': [0.6060606060606061, 0.13986013986013984,
                                        0.0790513833992095,
                                        0.05509641873278237],
                         'svc__kernel': ['rbf', 'poly', 'sigmoid', 'linear']},
             return_train_score=True, scoring='f1_weighted', verbose=True)
```

Si stampano i risultati della GridSearchCV:

```
df_results = pd.DataFrame(svm_gs.cv_results_)

display(df_results.sort_values(['rank_test_score'], ascending=True))
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_svc__C	param_svc__gamma	param_svc__kernel	params	split0_test_score	mean_test_score	std_test_score	rank_test_score
32	0.113696	0.0	0.422938	0.0	1	0.606061	rbf	{'svc__C': 1, 'svc__gamma': 0.6060606060606061...	0.616671	0.616671	0.0	1
48	0.117051	0.0	0.425911	0.0	2	0.606061	rbf	{'svc__C': 2, 'svc__gamma': 0.6060606060606061...	0.605966	0.605966	0.0	2
16	0.112698	0.0	0.441852	0.0	0.5	0.606061	rbf	{'svc__C': 0.5, 'svc__gamma': 0.606060606060606...	0.601473	0.601473	0.0	3
64	0.126148	0.0	0.438763	0.0	4	0.606061	rbf	{'svc__C': 4, 'svc__gamma': 0.6060606060606061...	0.600483	0.600483	0.0	4
80	0.132645	0.0	0.433921	0.0	8	0.606061	rbf	{'svc__C': 8, 'svc__gamma': 0.6060606060606061...	0.596164	0.596164	0.0	5

Anche in questo caso l' SVM con kernel gaussiano ha outperformato le altre SVM.

Riaddestrando la migliore SVM e testandola si ottengono i seguenti risultati:

	Accuracy	Precision	Recall	F1
training	0.916270	0.920993	0.916270	0.916047
validation	0.617347	0.616553	0.617347	0.616671
test	0.610453	0.611128	0.610453	0.608085

Matrice di Confusione No Normalizzazione:

	classe 0	classe 1	classe 2
classe 0	594	182	40
classe 1	311	617	188
classe 2	56	177	284

Matrice di Confusione Normalizzata rispetto le Vere Classi:

	classe 0	classe 1	classe 2
classe 0	0.727941	0.223039	0.049020
classe 1	0.278674	0.552867	0.168459
classe 2	0.108317	0.342360	0.549323

Matrice di Confusione Normalizzata rispetto le Classi Predette:

	classe 0	classe 1	classe 2
classe 0	0.618106	0.186475	0.078125
classe 1	0.323621	0.632172	0.367188
classe 2	0.058273	0.181352	0.554688

♣ **One-VS-Rest Classifier**: per implementare il secondo metodo si è utilizzato l' OneVsRestClassifier di scikit-learn che permette di eseguire l' **estimator** scelto (l' SVM nel nostro caso) utilizzando la strategia OvR.

```
n_features = X.shape[1]

C_list = [2 ** i for i in range(-2, 4)]

gamma_list = [1 / (i * n_features) for i in np.arange(0.15, 1.75, 0.5)]
ker_list = ['rbf', 'poly', 'sigmoid', 'linear']

model_to_set = OneVsRestClassifier(SVC())

pipe = Pipeline([('scaler', StandardScaler()), ('svc', model_to_set)])
hparameters = {'svc__estimator__kernel':ker_list, 'svc__estimator__C':C_list, 'svc__estimator__gamma':gamma_list}

svm_gs = GridSearchCV(pipe,
                      param_grid=hparameters,
                      scoring='f1_weighted',
                      return_train_score=True,
                      cv=zip([ind_train], [ind_val]),
                      verbose=True)
svm_gs.fit(X, y)
```

```
Fitting 1 folds for each of 96 candidates, totalling 96 fits
GridSearchCV(cv=<zip object at 0x000002581E9C40C0>,
             estimator=Pipeline(steps=[('scaler', StandardScaler()),
                                       ('svc',
                                        OneVsRestClassifier(estimator=SVC()))]),
             param_grid={'svc__estimator__C': [0.25, 0.5, 1, 2, 4, 8],
                         'svc__estimator__gamma': [0.6060606060606061,
                                                   0.13986013986013984,
                                                   0.0790513833992095,
                                                   0.05509641873278237],
                         'svc__estimator__kernel': ['rbf', 'poly', 'sigmoid',
                                                    'linear']},
             return_train_score=True, scoring='f1_weighted', verbose=True)
```

Si stampano i risultati della GridSearchCV:

```
df_results = pd.DataFrame(svm_gs.cv_results_)

display(df_results.sort_values(['rank_test_score'], ascending=True))
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_svc__estimator__C	param_svc__estimator__gamma	param_svc__estimator__kernel	params	split0_test_score	mean_test_score	std_test_score	rank_test_score
32	0.204808	0.0	1.085059	0.0	1	0.606061	rbf	{'svc__estimator__C': 1, 'svc__estimator__gamm...	0.620324	0.620324	0.0	1
16	0.211455	0.0	1.081198	0.0	0.5	0.606061	rbf	{'svc__estimator__C': 0.5, 'svc__estimator__ga...	0.613182	0.613182	0.0	2
48	0.235225	0.0	1.023042	0.0	2	0.606061	rbf	{'svc__estimator__C': 2, 'svc__estimator__gamm...	0.610475	0.610475	0.0	3
56	0.153214	0.0	0.756922	0.0	2	0.079051	rbf	{'svc__estimator__C': 2, 'svc__estimator__gamm...	0.605043	0.605043	0.0	4
0	0.203774	0.0	1.067730	0.0	0.25	0.606061	rbf	{'svc__estimator__C': 0.25, 'svc__estimator__g...	0.604446	0.604446	0.0	5

Riaddestrando la migliore SVM e testandola si ottengono i seguenti risultati:

	Accuracy	Precision	Recall	F1
training	0.946903	0.946952	0.946903	0.946892
validation	0.621429	0.626241	0.621429	0.620324
test	0.618620	0.618431	0.618620	0.615080

Matrice di Confusione No Normalizzazione:

	classe 0	classe 1	classe 2
classe 0	559	231	26
classe 1	274	724	118
classe 2	52	233	232

Matrice di Confusione Normalizzata rispetto le Vere Classi:

	classe 0	classe 1	classe 2
classe 0	0.685049	0.283088	0.031863
classe 1	0.245520	0.648746	0.105735
classe 2	0.100580	0.450677	0.448743

Matrice di Confusione Normalizzata rispetto le Classi Predette:

	classe 0	classe 1	classe 2
classe 0	0.631638	0.194444	0.069149
classe 1	0.309605	0.609428	0.313830
classe 2	0.058757	0.196128	0.617021

Come possiamo vedere il metodo One-VS-Rest, (Accuracy = 0.618620, Precision = 0.618431, F1 = 0.615080), si è comportato leggermente meglio del metodo One-VS-One, (Accuracy = 0.610453, Precision = 0.611128, F1 = 0.608085), ma la differenza è minima.

♦ **MULTI LAYER PERCEPTRON (MLP)**: diversamente da quanto fatto con la SVM per la rete neurale si può cercare di costruire un "training + validation" set "bilanciato" cioè con una presenza equivalente fra le varie classi.

Per far ciò si utilizza il seguente procedimento:

1) Si calcola il N_{min} , cioè il numero di campioni con la classe del dataset che ha una dimensione minore.
2) Si crea il training + validation set prendendo randomicamente $\frac{Nmin}{2}$ campioni da ogni classe.

A seguire si riporta l' esecuzione di tali passaggi:

```
X = white_wine_copy.iloc[:, :-1].values
y = white_wine_copy['quality'].values

y_nomi_classe = ['classe 0', 'classe 1', 'classe 2']

wine_counts = white_wine_copy['quality'].value_counts()
Nmin = wine_counts.min()
name_class_Nmin = wine_counts.index[wine_counts.argmin()]

display(pd.DataFrame({'numero classe': wine_counts.index,
                      'conteggio elementi': wine_counts.values}
                     )
       )

display(pd.DataFrame({'N_min': Nmin, 'name': name_class_Nmin}, index=[0]))

# Creazione nuovo training + validation set

Nmin_half = int(np.round(Nmin * 0.5))

indsX_trainval_list = []
indsX_test_list = []

for t in range(3):
    indsX_t = np.argwhere(y == t).flatten() # indici corrispondenti a immagini con target t
    indsX_trainval_t, indsX_test_t = train_test_split(indsX_t,
                                                      train_size=Nmin_half,
                                                      random_state=random_state,
                                                      shuffle=True)
    indsX_trainval_list.append(indsX_trainval_t)
    indsX_test_list.append(indsX_test_t)

indsX_trainval = np.concatenate(indsX_trainval_list, axis=0)
indsX_test = np.concatenate(indsX_test_list, axis=0)
# Mescoliamo (giusto per sicurezza) gli indici
np.random.shuffle(indsX_trainval)
np.random.shuffle(indsX_test)

X_trainval = X[indsX_trainval, :]
y_trainval = y[indsX_trainval]

X_test = X[indsX_test, :]
y_test = y[indsX_test]

display(pd.DataFrame({'trainval_size': len(y_trainval), 'test_size': len(y_test)}, index=[0]))
```

	numero classe	conteggio elementi
0	1	2198
1	0	1640
2	2	1060

	N_min	name
0	1060	2

	trainval_size	test_size
0	1590	3308

Si procede poi come per il problema binario applicando l' MLP classifier, dopo aver effettuato lo scaling, e si ottengono i seguenti risultati:

	Accuracy	Precision	Recall	F1
train. + val.	0.827673	0.829054	0.827673	0.826116
test	0.557739	0.593401	0.557739	0.553784

Matrice di Confusione No Normalizzazione:

	classe 0	classe 1	classe 2
classe 0	796	224	90
classe 1	531	702	435
classe 2	58	125	347

Matrice di Confusione Normalizzata rispetto le Vere Classi:

	classe 0	classe 1	classe 2
classe 0	0.717117	0.201802	0.081081
classe 1	0.318345	0.420863	0.260791
classe 2	0.109434	0.235849	0.654717

Matrice di Confusione Normalizzata rispetto le Classi Predette:

	classe 0	classe 1	classe 2
classe 0	0.574729	0.213130	0.103211
classe 1	0.383394	0.667935	0.498853
classe 2	0.041877	0.118934	0.397936

♦ **CONCLUSIONI:** come era prevedibile, diversamente dal problema binario, i risultati del problema multiclasse non sono stati molto incoraggianti.

Inoltre, a differenza del caso binario, l' SVM tramite GridSearchCV ha ottenuto una accuracy pari a 0.619, rivelandosi più efficiente della rete neurale che si è fermata a 0.558.

I vari accorgimenti effettuati non sono stati quindi sufficienti, si può infatti vedere come per entrambi i due classifier si è verificato un forte **overfitting,** riportando cioè una forte discrepanza tra l' accuracy del training set e quella del test set.

Ciò significa che i due algoritmi si sono allenati abbastanza sull' insieme selezionato, ma non sono molto in grado di fare previsioni corrette su insiemi sconosciuti.