REALIZZAZIONE DI UN CLASSIFICATORE PER IL DATASET *FIRST ORDER THEOREM PROVING*

Matteo D'Eramo

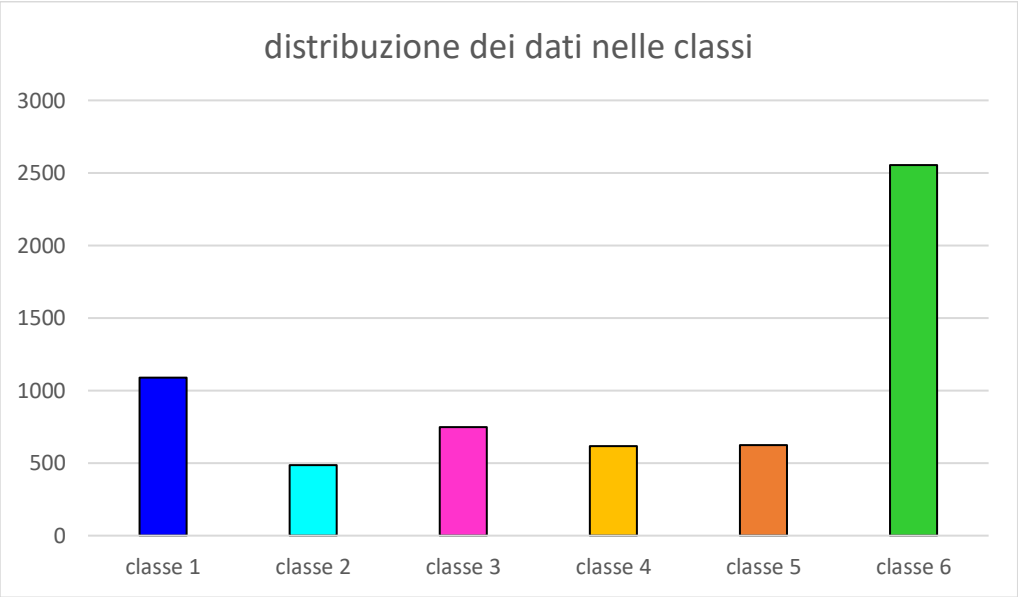
Matricola: 327994

1. DESCRIZIONE DEL PROBLEMA

La logica del prim'ordine è una teoria formale con cui è possibile esprimere enunciati e dedurre le loro conseguenze logiche in modo del tutto formale e meccanico; secondo questa teoria un teorema può essere inteso come il risultato di in una *formal proof*, ovvero una sequenza di sentenze le quali possono essere assiomi, assunzioni, formule o clausole. Questa sequenza può essere interpretata da un *theorem prover*, il quale verifica la correttezza di tutte formule e determina la veridicità del teorema. Un *prover* può utilizzare vari approcci euristici per la risoluzione delle *proof* i quali potrebbero comportare costi troppo onerosi in termini sia di spazio occupato che di tempo di calcolo.

L'obiettivo di questo progetto consiste nello studio ed implementazione in *python* di un classificatore per il dataset *First order theorem proving*: in questo *dataset* sono riportati una serie di istanze relative a teoremi, definiti da un insieme di 51 *features* numeriche. I teoremi sono classificati con 5 classi le quali rappresentano 5 differenti algoritmi euristici utilizzati da un *prover*: un qualsiasi teorema viene classificato con una di queste classi in base al minor tempo di calcolo impiegato per la risoluzione di tale problema (o in altri termini, la *fastest proof* ottenuta con i 5 algoritmi determina la classe di appartenenza). A queste classi è stata necessariamente aggiunta una sesta che rappresenta il caso in cui nessuno degli algoritmi euristici è in grado di risolvere il teorema.

Più nello specifico, il dataset contiene 6118 istanze composte ognuna da 51 *features* più 6 valori relativi alle *labels*, assegnati alle classi con valori +1 o -1. Il dataset è stato preventivamente suddiviso in *training set* (3059 istanze), *validation set* (1529 istanze) e *test set* (1530 istanze); per il progetto in questione questa suddivisione è stata ignorata e i dati sono stati riuniti per poi essere suddivisi secondo una differente logica. La distribuzione dei dati nelle classi è riportata nella tabella seguente:



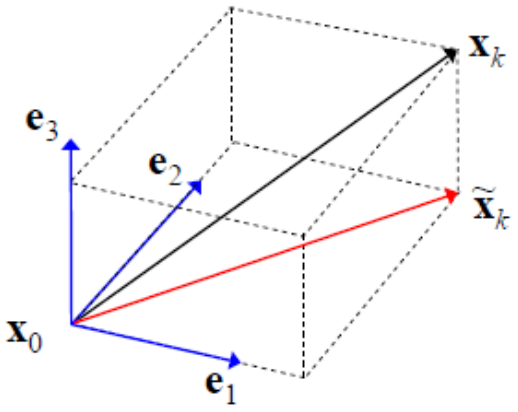
2. EXPLORATORY DATA ANALYSIS E CONSIDERAZIONI SUI DATI

Per poter leggere i *dataset* utilizziamo il metodo *read_csv* relativo alla libreria *numpy*: una volta caricati possiamo separare le *feature* dalle *label* mediante il metodo *iloc* (considerando le giuste colonne). le *features* e *label* ottenute dai 3 set specificati in precedenza vengono riunite per poi effettuare la separazione tra *training* e *test* set mediante il metodo *train_test_split*, in cui imponiamo che il *test* set abbia dimensione pari al 20% dei dati totali. Da notare che ogni *label* di ogni dato è inizialmente composta da un vettore a 6 dimensioni con valori +1/-1 a seconda dell'appartenenza alla classe ma per poter effettuare *fitting* e predizioni con esse sono state convertite in un singolo valore rappresentante la classe (1 per la prima, 2 per la seconda ecc.).

Una operazione importante prima di effettuare *training* dei dati consiste nell'analisi dei dati in modo da poter fare considerazioni specifiche e identificare proprietà di cui possono godere. Un importante facilitazione fornita dal *dataset* è la preventiva standardizzazione dei dati: la standardizzazione è un'operazione per cui le *features* vengono trasformate in modo tale da caratterizzare i valori secondo una distribuzione gaussiana a valor medio nullo e varianza unitaria, secondo la trasformazione $x' = \frac{x-\mu}{\sigma}$. questa scelta è assolutamente giustificabile per via dei differenti e disomogenei *range* di valori delle *features* originali: variabili molto disomogenee possono compromettere la qualità della classificazione sotto aspetti differenti a seconda del particolare modello considerato. Di seguito sono riportati alcuni dati riguardanti le *features* prima e dopo la standardizzazione (contenuti nel file *bridge-holden-paulson-details.txt*):

feature	*Min*	*Max*	*Standardized Min*	Standardized Max
1	0	1	-1.1052	2.0094
2	0.0078125	1	-3.7356	0.83152
3	0	1	-0.98411	2.7381
4	0	1	-1.0652	2.6448
5	0	0	-1.2401	2.3662
6	0.00038153	1	-0.88058	7.1945
7	0	0.98214	-1.7638	1.4393
8	0	0.9966	-0.82637	32.27
9	1	244	-0.70995	15.495
10	1	39.07	-0.91358	10.654
11	1	86	-1.8597	8.7417
12	1	11	-0.39657	42.424
13	12	16214	-0.54978	23.342
14	4.4643	990.14	-3.4736	0.75143
15	0.14141	1	-0.31199	31.367

La sola visualizzazione dei dati numerici non suggerisce particolari spunti per poter fare considerazioni. Per poter fare deduzioni è sempre utile poter visualizzare graficamente la disposizione dei dati; a tal fine è stata utilizzata la *PCA*:

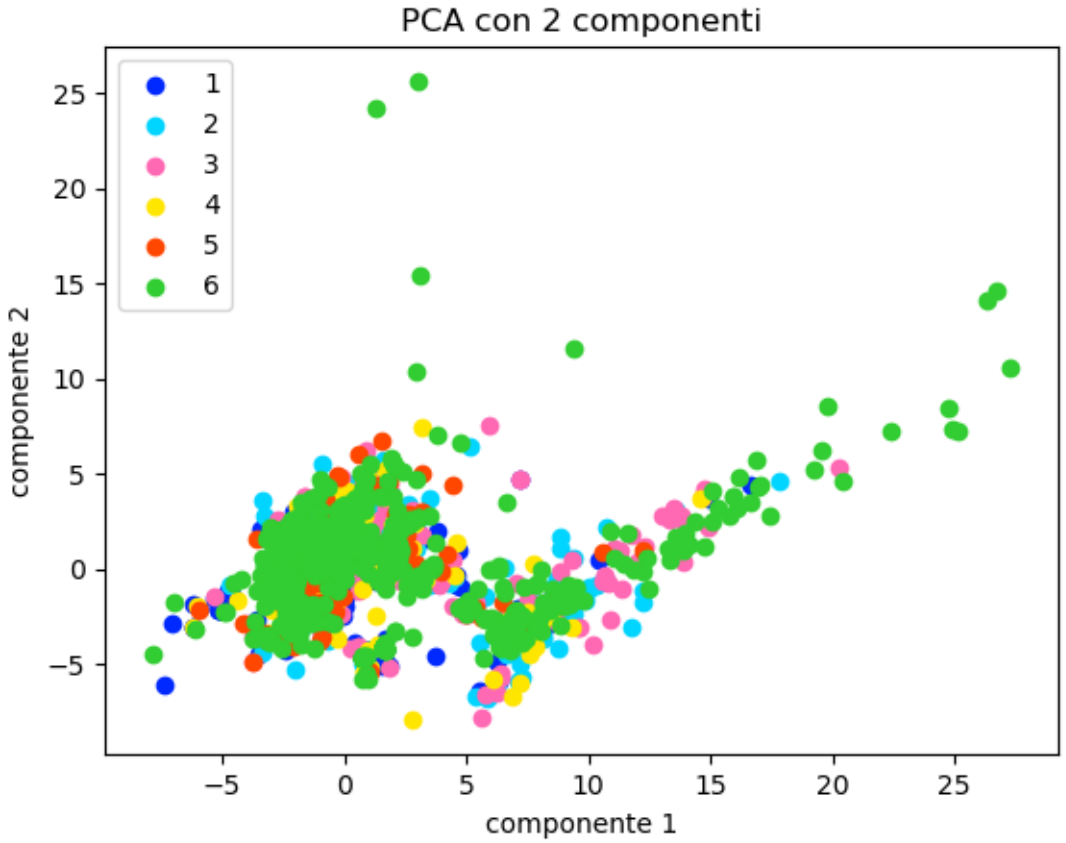


questo metodo non supervisionato (non tiene conto delle etichette dei dati) permette di effettuare una trasformazione delle *features* con il fine di rappresentare i dati secondo una nuova base con numero di componenti minore. Formalmente un qualsiasi dato x è esprimibile come combinazione lineare di una base a d componenti; l'obiettivo è considerare un numero $m < d$ componenti:

$$x_k = x_0 + \sum_{1}^{d} a_{ik} e_i \qquad x'_k = x_0 + \sum_{1}^{m} a_{ik} e_i$$

Il metodo posiziona la base nel punto medio dei dati e seleziona le *features* in base alle componenti con autovalori più grandi λ_i, i quali esprimono la varianza lungo la relativa componente e_i: questo significa che l'algoritmo seleziona le *features* in base alle componenti a maggior varianza.

Con la *PCA* riduciamo il numero delle *features* da 51 a 2 in modo da poterne dare una rappresentazione grafica (nella legenda i valori da 1 a 5 indicano le 5 euristiche); a livello implementativo è stata utilizzata la classe *PCA* assegnando il parametro $n\_components = 2$:



Con la sola rappresentazione grafica possiamo già affermare che i dati sono caratterizzati da scarsa separabilità, anche solo considerando ogni singola coppia di classi; per questo motivo difficilmente si potranno utilizzare modelli che effettuano una separazione lineare. Un ulteriore conferma è stata ottenuta provando a effettuare un *fitting* dei dati di training con una *support vector machine* con kernel lineare (con coefficiente $C = 1$, quindi il classificatore è approssimativamente ad *hard margin*), per poi eseguire la predizione sugli stessi dati: se fossero linearmente separabili l'errore sarebbe nullo e quindi $accuracy = 100\%$, cosa che non avviene dato che il risultato ottenuto è un modestissimo 29,28% (calcolata mediante *balanced_accuracy_score* che dà un valore più veritiero per classi sbilanciate). In virtù di questa condizione dovremmo utilizzare modelli adatti per situazioni non lineari.

Un altro aspetto analizzabile riguarda la correlazione tra le variabili: una correlazione alta può causare ridondanza e soprattutto il fenomeno della collinearità, per cui il classificatore sarà altamente sensibile a rumore e la qualità delle predizioni può essere molto instabile, oscillando tra risultati buoni e risultati pessimi. Per valutare la collinearità possiamo calcolare la matrice di correlazione, una matrice quadrata e simmetrica in cui ogni dimensione rappresenta il valore del coefficiente di correlazione tra una coppia di variabili. La matrice ha dimensione $51x51$ perciò è poco utile fornire una rappresentazione grafica (nel codice è comunque presente); per l'analisi dei valori sono stati utilizzati due cicli *for* per valutare ogni casella della matrice. I risultati di questa operazione hanno evidenziato come molte variabili siano caratterizzato da un elevato coefficiente di correlazione con altre: definendo un valore $\rho_{x,y} > 0{,}75$ come "alta correlazione positiva" e $\rho_{x,y} < -0{,}75$ come "alta correlazione negativa", sono state trovate 43 coppie di *features* soddisfacenti uno di questi due vincoli. Il risultato ottenuto risulta il primo esplicito campanello d'allarme per la qualità della predizione ed è quindi necessario trovare un criterio per poter gestire le *features* collineari; a questo proposito sono state ipotizzate due strategie che andremo a descrivere in dettaglio:

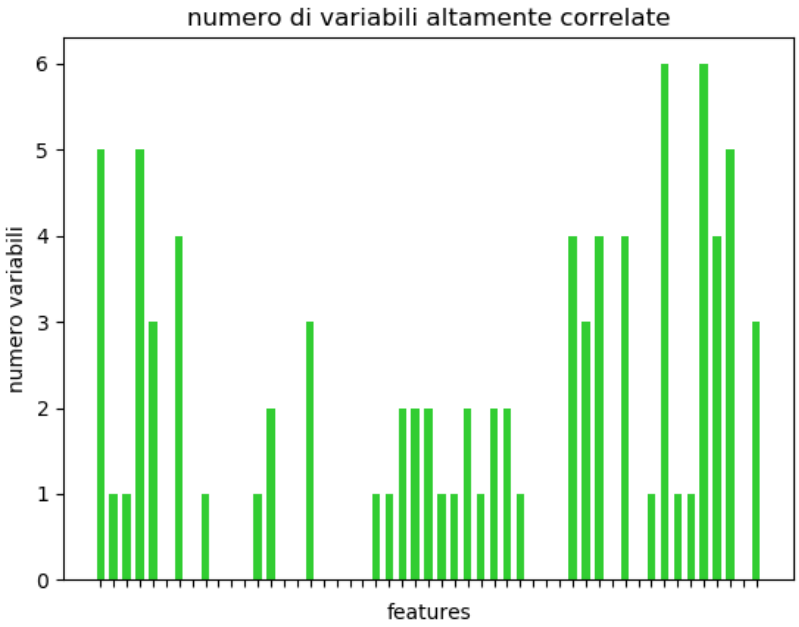
1. Rimozione diretta delle *features*
2. PCA dei dati

2.1 RIMOZIONE DELLE FEATURES

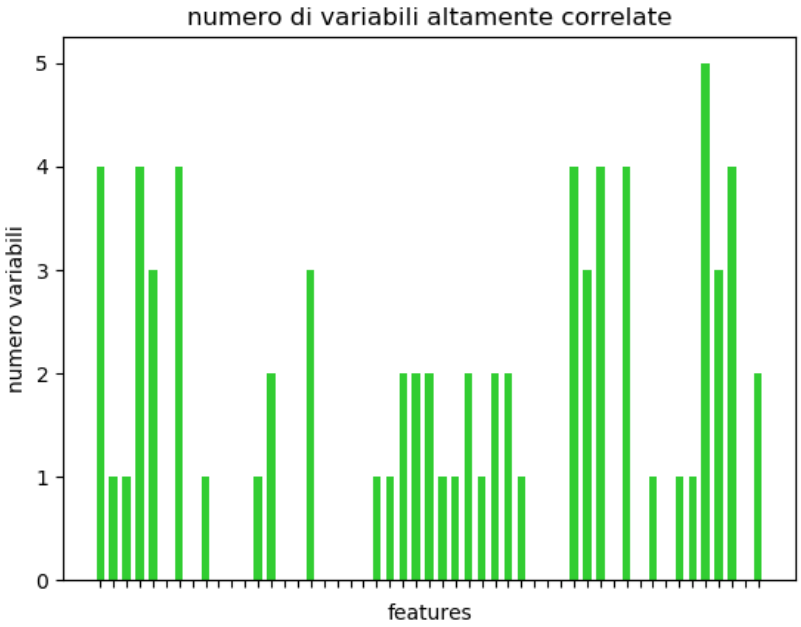
L'idea alla base di questo approccio è basata sulla rimozione delle *features* in maniera diretta selezionando quelle auspicabilmente meno utili o dannose per la predizione; considerando che la rimozione casuale di *features* tra quelle presenti nelle 43 coppie di variabili è privo di una parvenza di criterio e comporta un numero elevato di test, è necessario ipotizzare e tentare un approccio più specifico. Il metodo utilizzato per la selezione si basa sul fatto che singola variabile può essere altamente correlata con più di una variabile. Possiamo quindi utilizzare come metro di giudizio per valutare la criticità delle singole *features* il numero di variabili a cui ognuna è altamente correlata; Così facendo, rimuovendo una singola *feature* si può sperabilmente risolvere o attenuare il problema della collinearità per più variabili. A livello pratico è stato iterativamente trovato l'indice della *feature* presente nel maggior numero di coppie di variabili altamente correlate; una volta individuato il valore massimo, viene salvato l'indice in una lista e vengono azzerati tutti i coefficienti di correlazione relativi a quella variabile, in modo tale che al ciclo successivo non venga più

considerata. Per non rimuovere un numero eccessivo di variabili è stata imposta una soglia di 2 per il numero di coppie, valore per cui si interrompe l'operazione.

L'operazione è descritta dai seguenti grafici a barre, calcolati ad ogni iterazione: al primo ciclo il numero è possibile subito notare che due variabili (di indice 43 e 46) sono altamente correlate a ben 6 variabili; a parità di valori viene selezionata la prima variabile incontrata, perciò quella di indice 43 (corrispondente alla *feature n.44*).



Al secondo ciclo le correlazioni nella matrice relative alla *feature n.44* sono state azzerate in modo tale che non vengano prese in considerazione; di conseguenza calerà anche il numero di variabili le quali erano altamente correlate alla *feature* appena rimossa. In questa situazione la *feature* selezionata è quella di indice 46 (*feature n.47*):



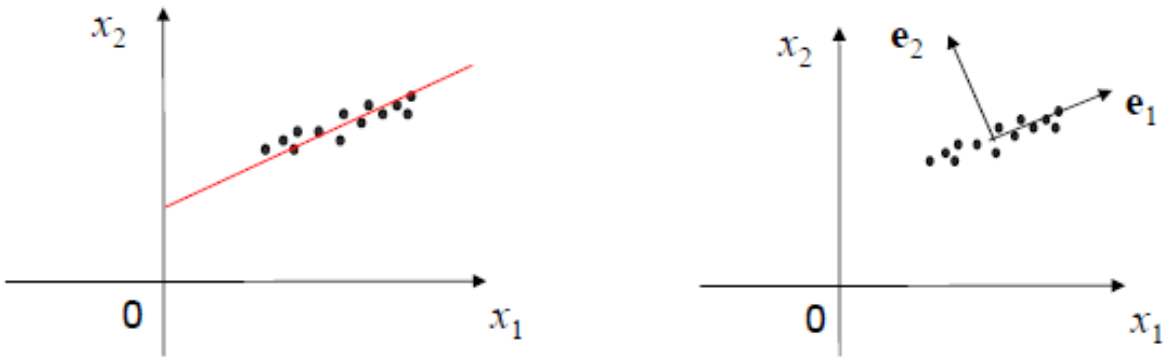
Il processo continua finché il massimo è pari a 2. A seguito di questo procedimento il vettore degli indici ottenuto è $[43, 46, 6, 36, 38]$; potrebbero però essere ancora presenti coppie di variabili con

valori di correlazione molto critici. Al tal fine sono state rivalutate le correlazioni della matrice per identificare eventuali valori elevati ($\rho_{x,y} > 0{,}9$ e $\rho_{x,y} < -0{,}9$); per ogni coppia di variabili trovata è stata selezionata una singola variabile in maniera arbitraria considerando quella con minore o maggiore varianza.

L'insieme di indici ottenuto sono $[43, 46, 6, 36, 38, 3, 16, 32, 50, 13]$ e $[43, 46, 6, 36, 38, 0, 40, 42, 48, 13]$ i quali verranno utilizzati per rimuovere le *features* (mediante il metodo *delete* di *numpy*).

2.2 PCA

Come già visto la PCA è una tecnica non supervisionata che consente di esprimere le *feature* secondo la migliore base di *m* componenti, in base alla varianza di queste ultime. Nel caso di variabili fortemente correlate è particolarmente utile perché permette di ridurre il numero di *feature* identificando le variabili altamente correlate con un'unica componente; in figura è mostrato un esempio esplicativo in 2 dimensioni in cui due variabili sono chiaramente correlate positivamente.



Nel nostro caso l'obiettivo sarà quello di individuare il numero di componenti ottimo in modo da ottenere la miglior trasformazione possibile dei dati; dato che la *PCA* verrà unita al modello utilizzato mediante una *pipeline*, la discussione relativa al numero di componenti è riportata nella prossima sezione.

È da sottolineare il fatto che la PCA è una trasformazione lineare e, nonostante possa essere adatto per *features* correlate, se ci fossero relazioni non lineari tra i dati questa trasformazione potrebbe non essere particolarmente utile.

3. MODEL SELECTION

La scelta del modello da utilizzare è un'operazione delicata che può fare la differenza per la predizione; prendiamo in considerazione i seguenti modelli:

- *Softmax regression*
- *Naive Bayes*
- *SVM*
- Rete neurale
- *K-NN*

Ed evidenziamone l'utilità nel problema corrente.

La *softmax regression* è un modello di separazione lineare, generalizzazione per il caso multiclasse della *logistic regression*; il fatto che sia una separazione lineare esclude a prescindere il suo utilizzo, dato che sicuramente non potrà convergere ad una soluzione in quanto come già visto in precedenza i dati non sono linearmente separabili. Il Naive Bayes è un algoritmo di tipo *generative*, per cui si intende stimare $p(x|y)$ e $p(y)$ in modo da poter calcolare la densità a posteriori $p(y|x)$ mediante la regola di Bayes; questo modello fa una assunzione molto forte per cui le *feature* sono definite come indipendenti dalle altre. Questa assunzione è generalmente di difficile rispettabilità e inoltre il fatto che le variabili siano molto correlate tra loro fa sì che esse siano tutt'altro che indipendenti, perciò anche questo approccio presumibilmente non sarà quello giusto per il problema corrente. La *support vector machine* è basato sull'identificazione della separazione a massimo margine ovvero più distante possibile dai dati; questo classificatore è però molto utilizzato anche per la realizzazione di regioni di separazioni non lineari mediante l'utilizzo di *kernel*, funzioni equivalenti ad un prodotto scalare:

$$K(x, z) = \varphi(x) * \varphi(z)$$

In cui φ è una trasformazione non lineare. L'*svm* è inoltre caratterizzato da un iperparametro C utilizzato per delineare la rigorosità della separazione: per $C \to 0$ il classificatore sarà ad *hard margin* e si otterranno molti errori e una separazione più rigorosa, mentre per $C \to \infty$ il classificatore sarà a *soft margin* e si otterranno pochi errori di classificazione ma la separazione sarà più blanda, con conseguente rischio di *overfitting*. Inoltre a seconda del *kernel* utilizzato saranno presenti uno o più iperparametri aggiuntivi che determineranno la complessità delle funzioni di separazione. Le reti neurali sono un set di algoritmi e tecniche utilizzate per l'apprendimento supervisionato: esse sono composte da unità (o nodi) arrangiate in strati successivi e collegate ad ogni altra unità dello strato successivo mediante delle connessioni pesate. Ogni unità di strati successivi al primo somma i valori delle connessioni e li trasforma mediante una funzione di attivazione che effettua una trasformazione matematica prima di passare il valore allo strato successivo. Una particolare rete neurale con funzione sigmoide (o altre) ed almeno tre livelli (un *input,* un *hidden* e *output* layer) è un *multilayer perceptron*; tra le proprietà di questa rete neurale c'è quella legata al teorema di approssimazione universale, per cui un *MLP* può approssimare qualsiasi funzione continua su un dominio di input compatto (a patto che ci siano un numero abbastanza grande di *hidden units*). Questo aspetto fa sì che il *MLP* sia adatto anche per separazioni non lineari dei dati. Infine il *K-NN* è il metodo supervisionato più semplice; questo si basa sull'assegnazione di un dato ad una classe in base ai *K* dati di *training* più vicini, assegnando la classe

in base alla più ricorrente tra i *K* dati. Questo algoritmo non utilizza funzioni di separazione ma si basa unicamente sulla distanza; con questo meccanismo può essere applicabile a situazioni lineari e non.

Nella precedente descrizione sono emersi 3 possibili modelli utilizzabili e che andremo a utilizzare nel problema corrente: *SVM, MLP* e *K-NN*. Da notare che questi 3 modelli (in particolare *SVM* e *K-NN*) sono particolarmente sensibili ai fattori di scala dei dati, perciò la standardizzazione è un'operazione che si addice al loro utilizzo. Vediamo ora le principali caratteristiche che andremo a modellare:

1. SVM

Un classificatore SVM può essere implementato in *python* mediante l'utilizzo di *sklearn.svm.SVC*; questo classificatore ha a disposizione numerosi parametri, la cui corretta regolazione influisce sensibilmente sulla qualità della classificazione. Nello specifico dobbiamo determinare:

- $\boldsymbol{C}$: parametro che determina la rigorosità del margine delle regioni di decisione
- $\boldsymbol{gamma}$**:** parametro legato alla complessità del *kernel* utilizzato
- $\boldsymbol{kernel}$**:** funzione-prodotto scalare utilizzata; tra i più noti ed utilizzati vi sono *linear, rbf* e *poly*
- $\boldsymbol{decision\_function\_shape}$**:** può essere *OVO* (*one vs one*) o *OVR* (*one vs rest*)

Per il problema corrente è stato imposto un *kernel rbf* (*radial basis function*) perché si basa su una combinazione lineare di funzioni radiali, la cui forma curvilinea si suppone possa essere adatta a questo tipo di dati (è stato utilizzato anche un *kernel* polinomiale (*poly*) ma i cui risultati sono stati nettamente inferiori a *rbf* e per questo motivo non è stato preso in considerazione).

2. *MLP*

Di questo classificatore (il quale può essere implementato mediante la classe di *scikitLearn MLPClassifier*) andremo a considerarne i seguenti parametri:

- $\boldsymbol{solver}$**:** solver utilizzato per l'ottimizzazione dei pesi
- $\boldsymbol{activation}$**:** funzione di attivazione dei nodi
- $\boldsymbol{alpha}$**:** parametro di regolarizzazione in norma L_2: un $alpha$ elevato penalizza valori dei pesi elevati, rendendo le regioni di decisione più blande. Al contrario un $alpha$ basso penalizza i pesi più bassi e determina regioni di decisione più definite, con rischio *overfitting*
- $\boldsymbol{hidden\_layer\_sizes}$**:** valore (x, y) in cui x indica il numero di unità per livello e y il numero di livelli (la profondità della rete).
- $\boldsymbol{early\_stopping}$**:** l'*early stopping* è una procedura di regolarizzazione per cui il *training* viene interrotto quando l'errore sul *validation* set è minimo. In pratica il *training* si interrompe quando non ci sono variazioni significative sull'errore dopo un tot di dati sottoposti.

utilizzeremo il solver *sgd (stochastic gradient descent*) in modo da poter regolarne i parametri di $learning\ rate$; il solver si occupa di minimizzare la funzione di costo, mediante la seguente relazione:

$$w^{(t+1)} = w^{(t)} - \eta \nabla \mathrm{E}(\mathrm{w}^{(\mathrm{t})})$$

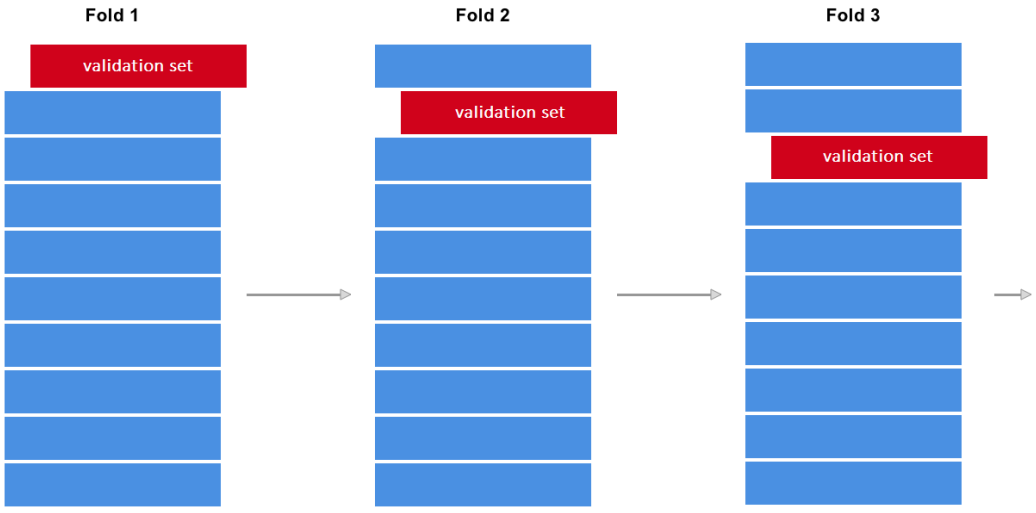
In cui η è il $learning\ rate$.

3. *K-NN*

Per questo modello andremo ad individuare i migliori valori dei seguenti parametri:

- $\boldsymbol{n\_neighbors}$**:** numero di "vicini" considerati
- $\boldsymbol{p}$**:** valore relativo alla famiglia di distanze di *Minkowsky:* per $p = 1$ avremo la *mahnattan distance* mentre per $p = 2$ la distanza euclidea
- $\boldsymbol{weights}$**:** funzione che associa i pesi ai punti; nel caso sia "*uniform"* tutti i punti in ogni *neighborhood* saranno equamente pesati, mentre se fosse "*distance"* i punti saranno pesati con l'inverso della loro distanza dal dato in questione

Spesso parte del set di dati di *training* può essere considerato come un cosiddetto "*validation set*": il *training* procede sul *training* set, dopodiché viene eseguita una valutazione sul *validation* set per fare il *tuning* degli iperparametri. tuttavia, partizionando i dati disponibili in tre set, riduciamo drasticamente il numero di campioni che possono essere utilizzati per l'apprendimento del modello e i risultati possono dipendere da una particolare scelta casuale per la coppia di set (*training, validation*). Una soluzione a questo inconveniente è la *cross validation,* un meccanismo di valutazione delle prestazioni di un classificatore. Il training set viene suddiviso in *k fold* di dimensione pari a $\frac{\dim\_trainingSet}{k}$ e tra questi ne viene selezionato uno che farà da *validation* set su cui andremo a testare il modello allenato con i restanti $k-1$ *fold*; questo processo viene eseguito iterativamente in modo che ogni *fold* sia utilizzato una ed una sola volta come *validation* set.



I *k* risultati ottenuti possono essere confrontati o ne può essere fatta una media per valutare le prestazioni medie del modello. La *cross validation* può essere eseguita mediante il metodo GridSearchCV, il quale confronta le prestazioni con tutte le combinazioni di parametri e seleziona gli

iperparametri che forniscono mediamente migliori prestazioni. Nelle ottimizzazioni è stato imposto un valore $k = 10$ per cui quindi verranno eseguiti 10 iterazioni per ogni combinazione di iperparametri.

Dal punto di vista dello *scoring* su cui ottimizzare gli iperparametri, l'*accuracy* è spesso una metrica utilizzata per la valutazione di un classificatore ma non è una misura veritiera nel caso di classi molto sbilanciate; infatti la predizione della classe più probabile (in questo caso la 6, che rappresenta il 40% delle etichette totali) può sbilanciare fortemente i risultati. Consideriamo ad esempio un dataset composto da 100 dati appartenenti a 2 classi, 90 nella prima e 10 nella seconda: ottimizzando rispetto l'accuracy potremmo ottenere un valore del 90%, ma essa sarebbe dovuta al fatto che, a causa del forte sbilanciamento, tutti i dati sono classificati con la prima classe e di conseguenza non ci sarebbe in pratica alcuna discriminazione tra classi.

Una misura più appropriata per questo tipo di problemi è l'*F1-score*, pari a:

$$F1 - score = \frac{2 * precision * recall}{precision + recall}$$

Questa metrica considera in maniera unica i valori di *precision* e *recall*, che misurano rispettivamente il *rate* di *false positive* e di *false negative*. L'ottimizzazione è stata eseguita per i tre modelli secondo tre modalità:

- *Features* originali
- *Features* ridotte (con entrambi i gruppi di indici)
- *Features* trasformate con *PCA*

Ai parametri elencati si aggiunge $n\_components$ nel caso venga utilizzata la PCA; questa potrà essere combinata con i modelli mediante una *pipeline* la quale sarà ottimizzata come modello unico. I valori ottenuti sono riportati in tabella.

SVM	C	$gamma$	$n\_components$
Original	100	0.2	-
Selected1	80	0.1	-
Selected2	100	0.1	-
trasformed	100	0.2	40

MLP	$activation$	$alpha$	$hidden\_layer\_sizes$	$early\_stopping$	$n\_components$
Original	*relu*	0,0001	(100,30)	*False*	-
Selected1	*tanh*	0.01	(100,30)	*False*	-
Selected2	*tanh*	0.01	(100,30)	*False*	-
trasformed	*relu*	0.01	(100,30)	*False*	40

K-NN	p	$n\_neighbors$	$weights$	$n\_components$
Original	1	10	*distance*	-
Selected1	1	5	*distance*	-
Selected2	1	5	*distance*	-
trasformed	2	5	*distance*	43

4. TEST

In questa fase sono stati testati i 3 modelli secondo le 4 varianti citate precedentemente; i valori di *F1_weighted* e *balanced_accuracy* sono riportati nella tabella seguente:

F1_weighted	*Original*	*Selected1*	*Selected2*	*transformed*
SVM	55.95%	**56.65%**	56.03%	55.78%
MLP	**53.65%**	52.89%	52.58%	53.14%
K-NN	58.6%	**59.05%**	59.04%	58.79%

Riguardo i risultati sperimentali possiamo fare le seguenti considerazioni:

- Il *SVM* ha ottenuto un incremento di prestazioni in entrambi i modelli con selezione delle *feature*, con un +0.7 nel caso di *Selected1*; per quanto riguarda la *PCA* vi è invece stato un decremento del valore di F1
- Il *MLP* ha ottenuto prestazioni migliori nella versione *original*, di conseguenza nessuna delle varianti ipotizzate ha apportato un incremento delle prestazioni; tuttavia lo *score* ottenuto con la *PCA* può essere considerato il migliore tra le varianti
- Per il *K-NN* ogni variante ha incrementato il valore di *F1* rispetto il caso *original*

Globalmente possiamo dire che la *feature selection* ha migliorati le prestazioni del *SVM* e *K-NN* in entrambe le varianti eseguite, segno che probabilmente è stata una mossa giusta; le *feature* non sono state selezionate secondo un criterio molto specifico perciò con un'analisi più dettagliata si potrebbe individuare il sottoinsieme ottimo di variabili da eliminare. Un aspetto da non sottovalutare per questi risultati è il ristretto numero di valori dei parametri considerati nelle ottimizzazioni con *GridSearchCV*; utilizzare un numero molto elevato di valori avrebbe aumentato esponenzialmente i tempi di calcolo (i quali sono già elevati con i valori presenti nel codice) e per questo motivo potrebbe non essere stato considerato un particolare valore che invece avrebbe fatto la differenza con una delle varianti dei modelli base, dato che il *tuning* corretto degli iperparametri è l'aspetto più importante per determinare buone prestazioni.

La *PCA* non ha dato i risultati sperati, segno che la trasformazione lineare delle *feature* non è stata efficace; possibili motivazioni di questo insuccesso possono riguardare il numero di componenti considerate per l'ottimizzazione oppure la presenza di relazioni non lineari tra i dati.