UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE E TECNOLOGIE

DIPARTIMENTO DI INFORMATICA
GIOVANNI DEGLI ANTONI

Corso di Laurea triennale in
Informatica

RICONOSCIMENTO DI VOLTI TRAMITE INSIEMI FUZZY

Relatore: Prof. Dario Malchiodi
Correlatore: Prof. Anna Maria Zanaboni

Tesi di Laurea di:
Tommaso Amadori
Matr. Nr. 892859

ANNO ACCADEMICO 2018-2019

Indice

Capitolo 1

Apprendimento automatico

Il Machine Learning (ML) insegna ai computer a compiere attività in modo naturale come gli esseri umani o gli animali: imparando dall'esperienza. In sostanza, gli algoritmi di ML usano metodi matematico-computazionali per apprendere informazioni direttamente dai dati, senza modelli matematici ed equazioni predeterminate. Gli algoritmi di ML migliorano le loro prestazioni in modo "adattivo" mano a mano che gli "esempi" da cui apprendere aumentano. Cerchiamo allora di capire cos'è il ML, come funziona e quali sono le sue applicazioni.

1.1 Approcci

I tipi di algoritmo del ML differiscono nel loro approccio, nel tipo di dati che utilizzano, che producono e nel tipo di attività o di problema che devono risolvere. Il ML può generalmente essere suddiviso in tre macro categorie:

- supervisionato,
- non supervisionato,
- apprendimento con rinforzo.

A queste si aggiunge solitamente una quarta categoria denominata semi-supervisionato.

1.2 Supervisionato

L'approccio supervisionato (o *supervised*) è una tecnica che prevede di lavorare su un insieme di dati associati ad etichette definite dall'utente. Le etichette sono delle classi (o gruppi) entro le quali sono suddivisi i dati. Sapendo che ogni dato è associato a un'etichetta si vuole arrivare a cogliere la relazione che vi è tra dati

ed etichette, così da poter predire le etichette a partire dai dati, anche lavorando con dati non visti durante la fase di apprendimento. Questa tecnica può fornire due diversi tipi di risultati: discreti o continui. Per comprendere meglio questo concetto proviamo a fare un esempio.

Consideriamo delle diagnosi mediche fatte su una serie di pazienti. Analizzando le diagnosi un medico è in grado di definire se il paziente è in salute o meno. Da qui possiamo estrapolare quindi due etichette differenti per il nostro caso: "in salute" e "malato". Fornendo come input a un classificatore questo insieme di dati con le rispettive etichette appena definite, il calcolatore, tramite un'algoritmo supervisionato, sarà in grado di fornire delle predizioni sulla possibile etichetta da attribuire ad ogni nuova diagnosi.

E' importante che vi sia a disposizione una quantità consistente di dati in input. In altre parole, è importante che vi siano molti dati da analizzare, perché quando questo non succede la capacità di predizione del sistema che si ottiene è spesso di scarsa qualità.

In questo esempio abbiamo utilizzato solamente le classi "in salute" e "malato", ma nulla ci vieta di definirne una terza o una quarta. Ad esempio possiamo etichettare un paziente come "asmatico" o "diabetico". Nel caso in cui non vogliamo avere una classificazione della salute del paziente, ma vogliamo quantificare l'aspettativa di vita, non è più possibile ricorrere a dei classificatori. Da qui nasce la necessità di passare da un valore discreto ad un valore continuo, pertanto si utilizza un modello diverso ovvero i regressori che costituiscono un modello per predire valori continui. Ad esempio, potremmo voler quantificare, data una specifica diagnosi, il tempo di guarigione per un paziente malato che per definizione si definisce su una scala di numeri continua.

I modelli di ML supervisionati hanno l'obiettivo di eseguire, con la maggior precisione possibile, una predizione su dati nuovi, mai visti prima. Per raggiungere questo scopo dobbiamo assicurarci che il modello produca stati lontani sia dal sovra-adattamento (*overfitting*) che dal sotto-adattamento (*underfitting*).

L'*overfitting* si verifica quando il modello tende ad adattarsi in maniera eccessiva ai dati che gli sono stati forniti per allenarsi, non permettendo la generalizzazione a nuovi insiemi di dati. L'*underfitting* invece, si verifica nel caso contrario dell'*overfitting*, quando il modello si basa su schemi troppo semplici e poco robusti, il che comporta la definizione di regole con scarsa qualità per la predizione di nuovi elementi.

Per esemplificare prendiamo spunto da un caso riportato in [1]. Basandoci sulla Tabella 1 vediamo quali sono i problemi generabili dall'*overfitting* e dall'*underfitting*. Supponiamo di voler predire se un cliente vorrà acquistare una barca. Osservando attentamente la tabella si può notare che applicando la regola: "Se un cliente ha più di 45 anni, ha meno di 3 figli o non è divorziato, allora vorrà comprare una

Tabella 1: Statistica per determinare l'acquisto di una nuova barca

Età	Macchine	Case possedute	Figli	Stato civile	Barca
66	1	Sì	2	Vedova	Sì
52	2	Sì	3	Sposato	Sì
22	0	No	0	Sposato	No
25	1	No	1	Single	No
44	0	No	2	Divorziato	No
39	1	Sì	2	Sposato	No
26	1	No	2	Single	No
40	3	Sì	1	Sposato	No
53	2	Sì	2	Divorziato	Sì
64	2	Sì	3	Divorziato	No
58	2	Sì	2	Sposato	Sì
33	1	No	1	Single	No

barca", tutte le predizioni (su questo dataset) saranno corrette.

Ma questo significherebbe anche che se in futuro un cliente C volesse comprare una barca e non rispettasse la regola sopra definita (magari perché ha semplicemente 44 anni o perché ha 4 figli), la predizione del sistema sarebbe "C non vuole comprare una barca", quindi errata. Questo è il caso dell'*overfitting*: stabilire le regole di predizione su troppi dati, in maniera troppo rigida.

Lo stesso si può fare al contrario, ossia quando le regole di predizione adottate dal modello si basano su pochi dati e/o le regole sono troppo vaghe. Supponiamo che il sistema identifichi un cliente come possibile acquirente di una barca se segue la seguente regola: "Se un cliente ha una casa allora vorrà comprare una barca". È naturale, leggendo la regola, pensare che questa sia eccessivamente generica. Questo è il caso dell'*underfitting*, ossia il caso in cui si definisce un modello che segue regole troppo vaghe, regole che portano a una scarsa qualità di predizione.

Quindi quello che vogliamo trovare è un modello che si posizioni a metà tra l'*overfitting* e l'*underiffing*. La complessità del modello, ossia quante e quali variabili vanno considerate, è un importante aspetto da considerare. Con un modello troppo "semplice" si rischia di utilizzare dati poco significativi per effettuare predizioni. Al contrario, quando è troppo complesso, rischiamo di utilizzare troppe variabili che non permettono di focalizzarci su quelle che sono davvero significative, il che può compromettere la performance del modello. In Figura 1 è possibile osservare quanto appena detto. L'asse orizzontale rappresenta la complessità del modello mentre quella verticale rappresenta l'accuratezza della predizione effettuata dello stesso. La curva blu raffigura l'accuratezza della predizione sui dati di allenamento e, analogamente a questa, la curva verde rappresenta l'accuratezza sui

dati che non sono stati usati durante la fase di allenamento. Nel caso di una scarsa complessità del modello avremo un'accuratezza di predizione bassa in entrambi i casi. Con il crescere della complessità del modello (e quindi con il crescere della varietà di dati da poter utilizzare) notiamo che la curva blu aumenta il suo livello di precisione, mentre quella verde raggiunge il picco quando si trova ad un giusto compromesso di complessità, superato il quale torna a decrescere. Il picco di cui parliamo è lo *Sweet spot*, cioè il punto che rappresenta il miglior compromesso tra precisione nella predizione e complessità del modello in caso di dati mai visti. La curva verde decresce quando la complessità diventa eccessiva. Con un numero eccessivo di variabili da considerare si rischia infatti di far perdere rilievo alle variabili più importanti tra tutte quelle considerate.

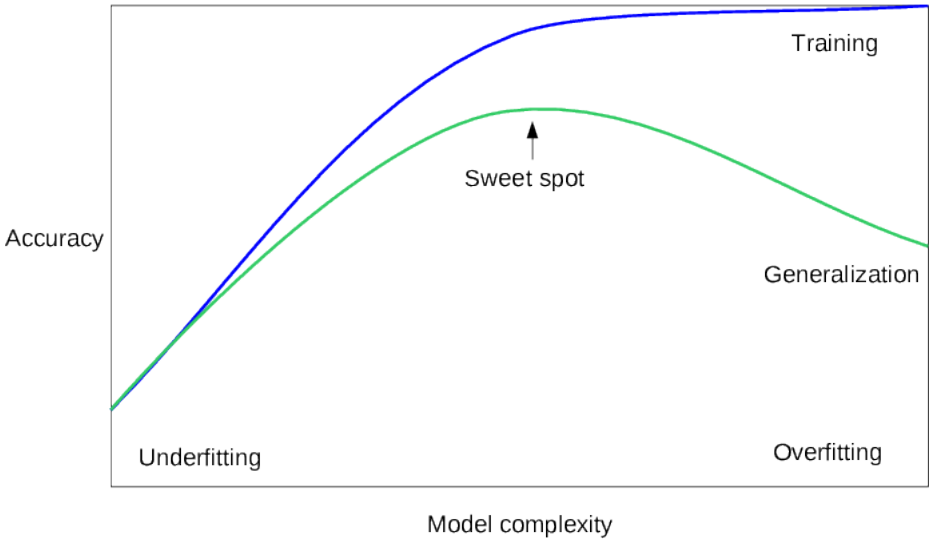


Figura 1: Trade-off tra overfitting e underfitting

1.2.1 k-Nearest Neighbor

Vediamo nello specifico uno dei più semplici algoritmi di ML: *k-Nearest Neighbor* (k-NN). Questo è un algoritmo utilizzato sia per la classificazione che per la regressione. In entrambi i casi l'algoritmo si basa sul parametro k fissato. Esso definisce il numero di vicini da prendere in considerazione per fare la predizione.

Supponiamo di avere due *feature* (o caratteristiche), per semplicità, feature_0 e feature_1 le quale descriveranno – insieme alla classe – ogni record del nostro dataset. Nel caso di classificazione tramite il k-NN in Figura 2 è mostrato come un elemento non ancora etichettato venga classificato in base al tipo predominante

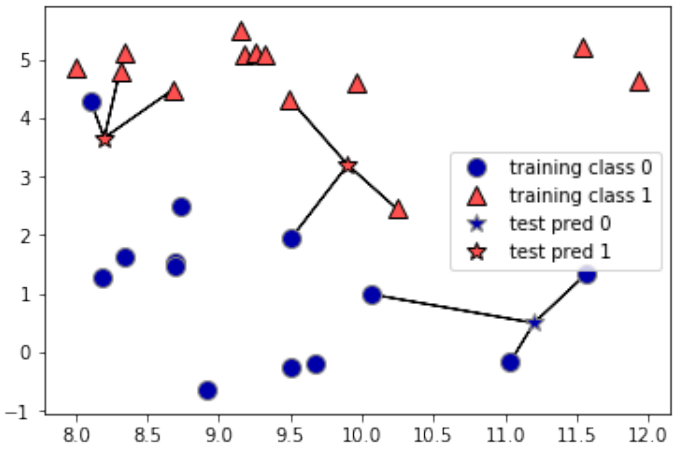


Figura 2: Classificazione tramite k-NN di 3 elementi. Le stelle rosse rappresentano una classificazione rispetto alla classe 1 (dei triangoli) mentre le stelle blu rappresentano una classificazione rispetto alla classe 0 (dei cerchi).

dei suoi vicini. La scelta di k è quindi l'unica, ma fondamentale, scelta da prendere per determinare la precisione nella predizione dei futuri elementi.

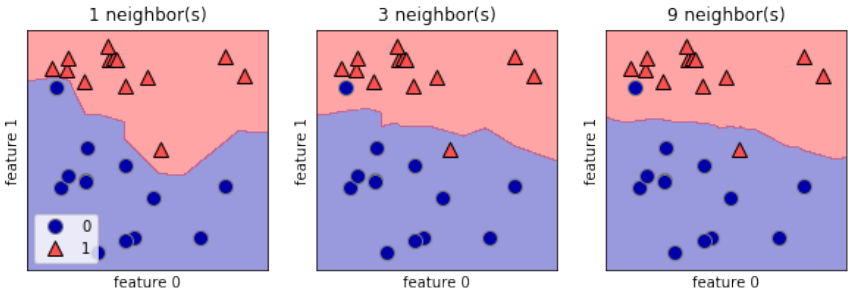


Figura 3: Risultati di un classificatore k-NN, per diversi valori del parametro k su un medesimo dataset. I cerchi e i triangoli indicano le osservazioni del dataset appartenenti a due classi, mentre le aree blu e rosse indicano gli esiti della classificazione.

In Figura 3 viene mostrato come influisce la scelta di differenti parametri k su uno stesso campione.

Questo algoritmo è spesso utilizzato con un k dispari per escludere casi di indecisione e quindi poter sempre definire la classe del nuovo dato.

Nel caso di regressione tramite il k-NN, il risultato sarà pari alla media dei valori target dei k più vicini. Immaginiamo di avere, nel nostro dataset, due *feature* per ogni elemento: Target e Feature. Feature rappresenta il valore su cui vogliamo

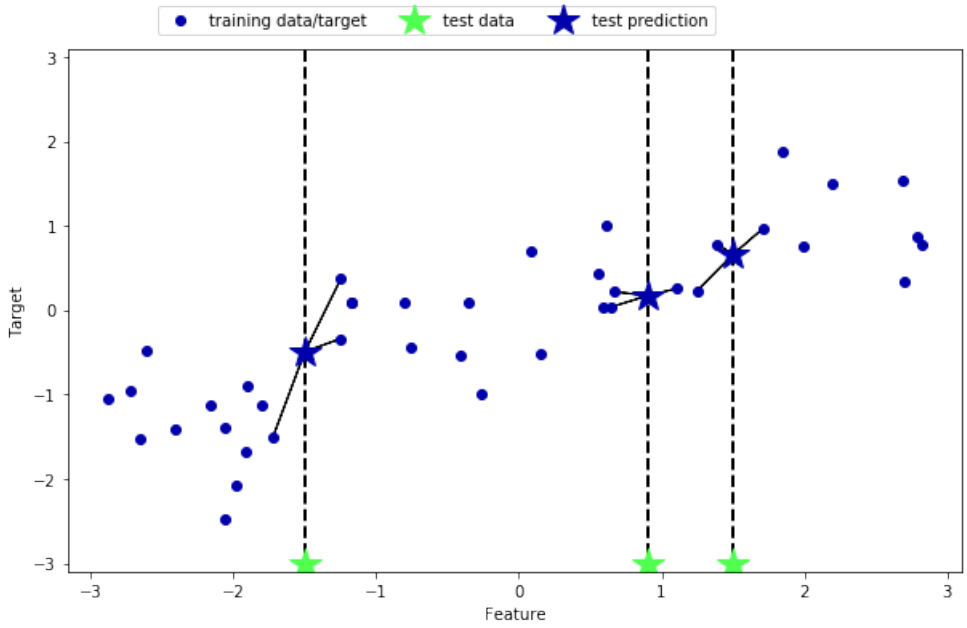


Figura 4: Risultato di un regressore k-NN, per k pari a 3. I cerchi indicano le osservazioni del dataset, le stelle verdi indicano il nuovo elemento da predire e le stelle blu gli esiti della predizione.

basare il modello e Target il valore che vogliamo predire. Prendendo ad esempio un k pari a 3 otteniamo (come vediamo in Figura 4) che il nuovo elemento da predire (la stella verde) avrà come valore Target (la stella blu) la media dei Target dei 3 elementi più vicini sull'asse delle ascisse (l'asse delle *feature*).

1.2.2 Modelli lineari

I modelli lineari sono una classe di modelli che cercano di effettuare predizioni utilizzando una funzione lineare basata sull'insieme delle *feature* dell'elemento da analizzare. Nel caso della regressione, la funzione di cui parliamo è definita come segue:

$$y = w_0x_0 + w_1x_1 + ... + w_nx_n + b,$$

dove n è il numero di *feature*, x_i le *feature*, w_i i pesi da attribuire a queste ultime e b un termine noto. Prendendo una sola *feature* (quindi n pari a 1), y risulterebbe:

$$y = w_0x_0 + b,$$

che è esattamente la funzione di una linea retta, dove w_0 è il coefficiente angolare e b è lo scostamento dall'origine degli assi.

Riprendendo l'esempio precedente, supponiamo che si voglia quantificare il numero dei giorni necessari per guarire un paziente malato. Supponiamo inoltre, per semplicità, di avere una sola caratteristica indicante l'età del paziente. Nel grafico in Figura 5 è rappresentato il dataset dei pazienti le cui coordinate sono l'età sull'asse delle ascisse e i giorni di guarigione sull'asse delle ordinate.

È possibile tracciare una retta denominata retta di regressione che approssima tutti i punti definiti nel dataset.

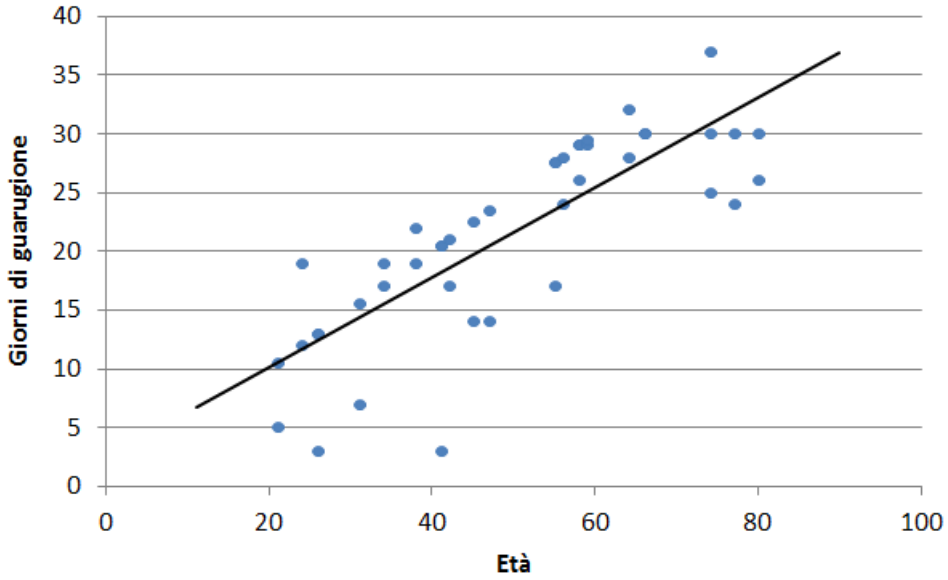


Figura 5: Regressione lineare. I punti rappresentano i pazienti e la retta nera rappresenta la retta di regressione che approssima meglio all'andamento di tutti i punti

I modelli lineare si possono applicare anche al contesto della classificazione, modificando leggermente la loro formulazione ovvero introducendo degli intervalli per definire a quale classe appartiene il singolo caso. Nella classificazione binaria ad esempio, la formula risulterebbe come segue:

$$y = w_0x_0 + w_1x_1 + ... + w_nx_n + b > 0,$$

dove, supponendo di avere le classi C_1 e C_0, se la y fosse maggiore di 0, l'oggetto descritto dagli x_i verrebbe classificato come C_1, altrimenti come C_0.

1.2.3 Support Vector Machine

Le Support Vector Machine (SVM) sono una classe di modelli che si occupa di individuare un iperpiano utile a separare i punti in uno spazio e quindi dividerli

in diversi gruppi.

Il primo problema che le SVM devono risolvere è capire quale sia l'iperpiano che suddivide nel modo "migliore" i dati.

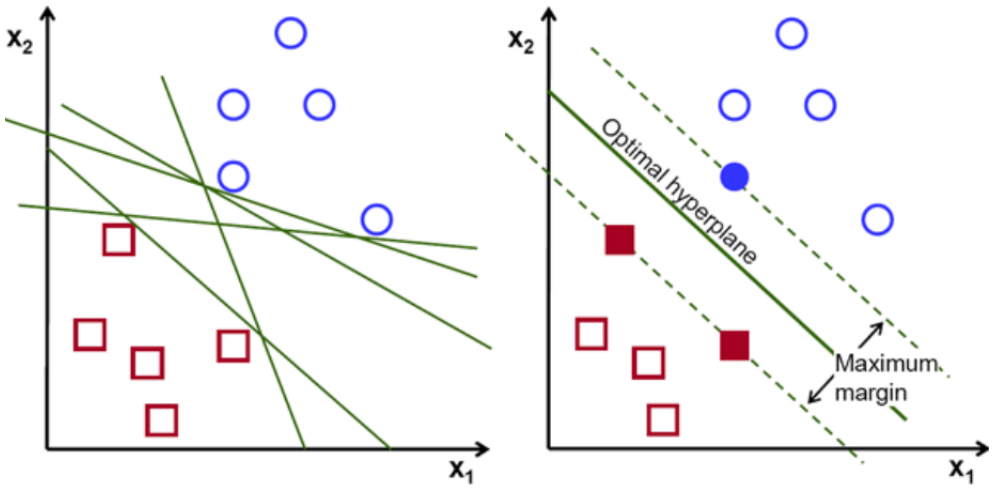


Figura 6: Possibili iperpiani che dividono lo spazio (a sinistra) e iperpiano "migliore" (a destra)

Nel grafico di sinistra della Figura 6 è possibile notare due gruppi distinti di punti (i cerchi blu e i quadrati rossi), che possono essere divisi dagli iperpiani raffigurati con linee verdi. Queste linee sono candidate ad essere i "migliori" divisori per i due insiemi ma per decidere quale sia il migliore si considerano i punti P più vicini a ogni iperpiano I. Questi punti vengono definiti vettori di supporto (o *support vector*). Per ogni iperpiano I e i relativi vettori di supporto V, viene calcolata la distanza tra I e V che viene chiamato "margine". Si definisce quindi iperpiano migliore, l'iperpiano che riesce a massimizzare il margine dai rispettivi vettori di supporto. Nell'immagine destra in Figura 6 mostra l'iperpiano "migliore" ovvero l'iperpiano che massimizza il "margine".

Nei casi reali capita spesso che lo spazio non è linearmente separabile. Per ovviare a questo problema si ricorre alle funzioni *kernel* che sono in grado di mappare dei vettori da uno spazio n-dimensionale a uno spazio m-dimensionale. Supponiamo il caso, rappresentato in Figura 7, dove si hanno due sole dimensioni ma dove non è possibile suddividere i punti con una semplice linea.

Tramite una funzione *kernel* trasformiamo i punti definiti dalle coordinate x e y, in punti aventi coordinate x, y e z. Questa trasformazione è conosciuta come trucco del *kernel* (o *kernel trick*). Essa permette di trasformare uno spazio n-dimensionale in uno spazio m-dimensionale, dove m è spesso più grande di n. Così facendo possiamo rappresentare l'insieme in uno spazio tridimensionale e tracciare

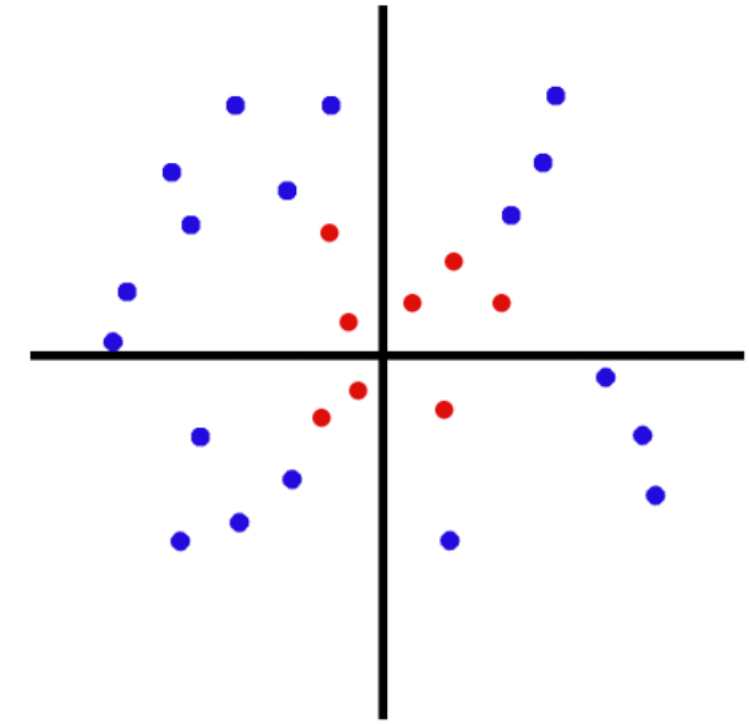


Figura 7: Dati non divisibili linearmente

un iperpiano che suddivide in modo lineare i punti nello spazio, per poi ridefinirlo secondo le due dimensioni iniziali di partenza. Aggiungiamo quindi una terza dimensione z e definiamola come segue:

$$z = x^2 + y^2.$$

Il risultato ottenuto è mostrato in Figura 8 e da esso si evince che si possono facilmente dividere i punti nelle due classi utilizzando una retta di equazione

$$z = k,$$

dove k è una costante.

Quindi quando ci si trova davanti a problemi di suddivisione dell'iperspazio bisogna spesso ricorrere al trucco del *kernel*.

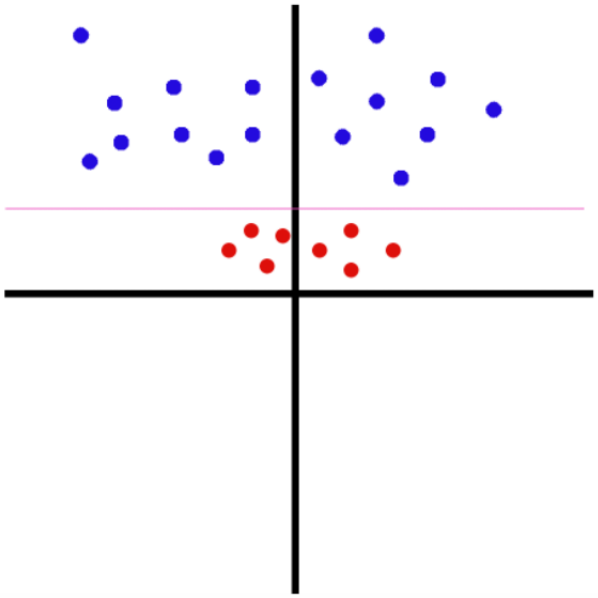


Figura 8: Dati visti sull'asse z

1.2.4 Alberi di decisione

Gli alberi di decisione, o *decision trees* (DT) sono dei modelli di classificazione o di regressione la cui logica si basa su una struttura ad albero. I nodi dell'albero rappresentano delle domande (o *test*) la cui risposta è di tipo binario (vero o falso) mentre le foglie rappresentano le classi che vogliamo predire.

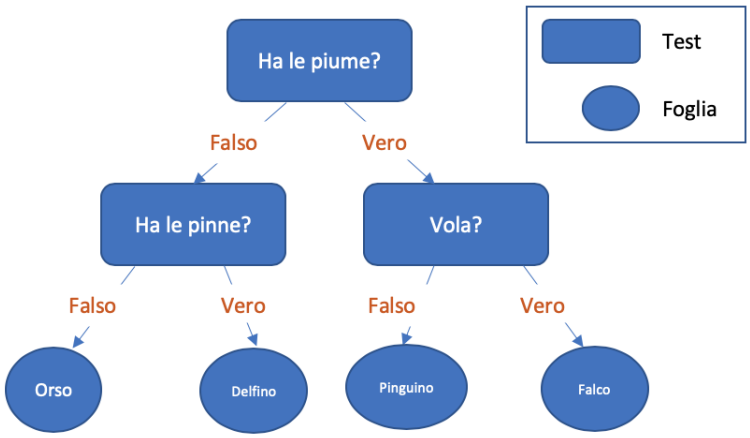


Figura 9: Esempio di albero di decisione per la classificazione di un animale

Supponiamo, ad esempio, di voler distinguere un animale tra: falco, pinguino, delfino e orso. Nell'esempio in Figura 9 l'algoritmo parte dalla domanda D: "Ha le piume?". In questo modo abbiamo definito il primo nodo N dell'albero rappresentato da D. Sapendo che i *test* previsti dall'algoritmo restituiscono un output binario, da N si diramano due sotto-alberi, rappresentati dalle categorie C_0("ha le piume") e C_1("non ha le piume"). Considerando gli elementi appartenenti a C_1 (orso e delfino), mediante un'ulteriore domanda si possono distinguere i due animali e quindi determinare la classe di appartenenza di ciascuno di essi. Seguendo quindi la domanda "Ha le pinne?", se la risposta è si, si sta parlando del delfino, ovvero l'unico animale tra i quattro che non ha le piume e ha le pinne, altrimenti deve trattarsi dell'orso. Seguendo questa logica è possibile arrivare – con le giuste domande – a fare delle predizioni.

Questo è un esempio eccessivamente semplificato. Nella realtà i dati che vengono analizzati hanno spesso valori di tipo continuo, quindi il *test* è del tipo: "x è maggiore di k?", dove x rappresenta una *feature* del dataset e k rappresenta una costante.

Una volta definiti i *test* è possibile suddividere lo spazio su cui rappresentiamo i dati in diverse aree, ognuna delle quali è rappresentata da una foglia (e quindi da una classe).

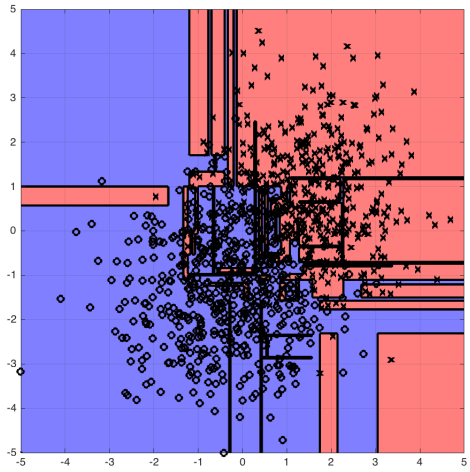


Figura 10: Overfitting dei dati con l'albero di decisione

Come abbiamo già visto negli altri algoritmi, il problema dell'*overfitting* e *underfitting* sono problemi ricorrenti che si presentano anche nel caso degli alberi di decisione. Infatti se viene costruito un albero troppo dettagliato, e quindi con un elevato livello di profondità, il modello tende ad adattarsi in maniera eccessiva ai dati usati in fase di allenamento generando un problema di *overfitting*. In Figura 10 è raffigurato un albero di decisione evidenziando le aree che corrispondono alle

sue foglie su un piano bidimensionale. Nella figura è possibile osservare zone blu e rosse eccessivamente piccole le quali sono sintomo di un elevato livello di dettaglio nelle domande. Siamo quindi in presenza di un problema di *overfitting*.

Analizzando un caso simile è possibile vedere tramite il grafico riportato in Figura 11 come l'errore su un insieme di dati non incluso in quello di allenamento cresce con l'aumentare della profondità. Il grafico mostra che più permettiamo all'albero di avere tanti livelli, più lui ha la capacità di adattarsi meglio ai dati di training, e a partire da una certa lunghezza inizia a farlo a detrimento della sua capacità di generalizzazione. Questo succede proprio perché il modello si è adattato in maniera eccessiva all'insieme di dati di allenamento.

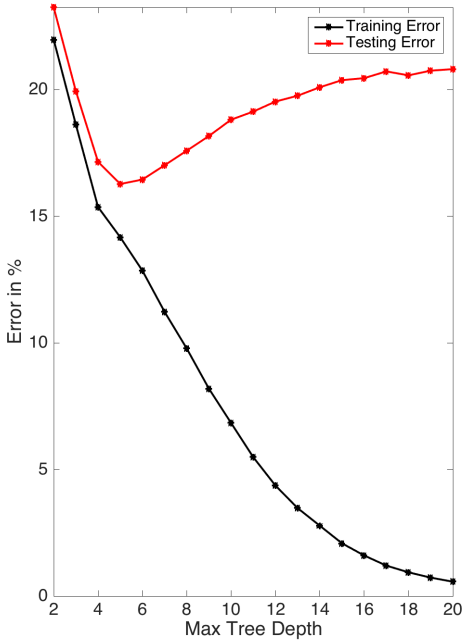


Figura 11: Grafico di *overfitting* per diversi alberi di decisione in cui ogni punto corrisponde ad un modello DT la cui profondità è definita sull'asse delle ascisse. La curva nera rappresenta l'errore di precisione nella predizione di dati di allenamento mentre la curva rossa rappresenta l'errore sui dati di test.

Per risolvere questo problema esistono due strategie:

- far terminare lo sviluppo dell'albero dopo pochi passi (pre-potatura) ovvero limitare a priori la profondità dell'albero,
- rimuovere i nodi che contengono informazioni poco significative (post-potatura).

1.3 Non supervisionato

La tecnica non supervisionata (o *unsupervised learning*) è il secondo importante approccio all'applicazione del ML. Cosa si intende dire con "non supervisionata"? Come può una macchina imparare se nessuno la guida nella scelta delle decisioni? Questa è proprio la sfida che si vuole superare con questa tecnica, ovvero far estrapolare al calcolatore delle informazioni "nascoste" all'interno dei dati che gli vengono forniti. Queste informazioni solitamente sono legami, schemi o regole che i dati tendono a seguire.

Ci sono diversi utilizzi di ML non supervisionato, in questa sezione ci limiteremo a elencarne alcuni. Il principale algoritmo di *Unsupervised Learning* è il clustering, ossia una tecnica in grado di suddividere in gruppi distinti elementi che hanno dati e caratteristiche in comune.

Supponiamo di aver scattato una serie di fotografie in cui sono raffigurate delle persone e decidiamo di caricarle su un social network. Supponiamo inoltre che durante il caricamento il social network su cui le stiamo caricando visiona le foto e applichi proprio un algoritmo di clustering. In che modo? L'algoritmo non sa né chi siano le persone raffigurate né quante siano. Andrà, però, a cercare tutti i volti nelle fotografie che abbiamo caricato e, successivamente, dopo aver definito una lista di tutti i volti presenti in ogni foto, tramite un algoritmo di clustering cercherà delle somiglianze in questi volti. Alla fine della sua esecuzione l'algoritmo avrà raggruppato le foto dove è presente lo stesso soggetto.

Un altro esempio di utilizzo ricade nell'ambito della sicurezza informatica. Al giorno d'oggi i tipi di attacco conosciuti sono probabilmente solo la punta dell'iceberg. Ricorrendo, però, a tecniche di clustering possiamo riuscire a individuare e bloccare attacchi tuttora sconosciuti. Supponiamo di essere collegati al sito della nostra banca che memorizza tutte le operazioni che facciamo. Supponiamo inoltre, di effettuare quotidianamente delle specifiche operazioni all'incirca alla stessa ora, collegandoci da una determinata località geografica. Tramite il ML non superivsionato queste informazioni potrebbero essere utilizzate per creare dei cluster, che ci individuino sulla base delle nostre operazioni. Supponiamo ora che un malintenzionato, dall'altra parte del mondo, a un orario differente da quello a noi abituale, riesca ad accedere al nostro profilo bancario. L'algoritmo sarebbe in grado di notare che è in atto qualcosa di anomalo. Nonostante nessuno abbia specificato al calcolatore quali sono le operazioni abituali effettuate dell'utente, il calcolatore, mediante tecniche di clustering, è in grado di riconoscere le operazioni abituali e mandare un messaggio di allarme se individua possibili casi anomali.

1.4 Semi-supervisionato

L'approccio semi-supervisionato (o *semi-supervised*), non è un vero e proprio approccio, bensì una tecnica che sta a metà tra le due appena viste: supervisionato e non supervisionato. Questo approccio consiste nel combinare le due tecniche e fornire un risultato basandosi su un input eterogeneo costituito da dati etichettati e dati non etichettati.

Questo metodo risulta utile quando si ha una grande mole di dati che vengono etichettati manualmente da utenti specializzati. Nella situazione reale non sempre questo è possibile, proprio perché possono mancare risorse umane competenti o tempistiche adeguate per etichettare tutti i dati. Esistono differenti algoritmi per l'apprendimento automatico mediante un sistema semi-supervisionato:

- *self training*,
- *multi-view training*,
- *self-ensembling*.

Per quanto riguarda il *multi-view training*, possiamo dire che esso mira a formare diversi modelli con diverse visualizzazioni dei dati. Idealmente queste viste sono complementari e i modelli possono collaborare per migliorare il risultato finale. Queste viste possono differire in diversi modi, ad esempio possono differire nelle funzionalità che utilizzano, nelle architetture dei modelli o nei dati su cui i modelli vengono formati.

Il *self-ensembling*, come il *multi-view training*, punta a combinare diverse varianti dei modelli. A differenza di quest'ultimo, però, la diversità nei modelli non è un punto chiave perché il *self-ensembling* utilizza principalmente un singolo modello in diverse configurazioni al fine di rendere più affidabili le previsioni del modello finale.

Vediamo ora più in dettaglio l'algoritmo di *self training* che è stato uno dei primi algoritmi a essere sviluppato ed è l'esempio più diretto di come le previsioni di un modello possono essere incorporate nel training del modello.

L'algoritmo di *self training* prevede, quindi, di basarsi per quanto possibile su dati che sono stati preventivamente etichettati, avendo anche a disposizione dati non etichettati. Questi ultimi vengono comunque utilizzati, ma in maniera più cauta. Prima di allenare il modello l'algoritmo si concentrerà ad etichettare gli input non ancora etichettati.

Come viene spiegato nell'articolo [12] la logica di classificazione dei dati non ancora classificati segue quanto scritto: "Formalmente, l'auto-etichettamento avviene su un modello M avente un insieme L di dati di allenamento etichettati con delle etichette contenute in C e un insieme non etichettato U. A ogni iterazione,

per ogni $x \in U$, il modello fornisce delle predizioni su x sotto forma di probabilità $p(x, c)$ ovvero la probabilità che x appartenga alla classe c per ogni $c \in C$. Tra le probabilità appena calcolate, definiamo $P(x, c)$ come la probabilità avente il valore maggiore, allora se P è più grande di una soglia T, x verrà aggiunto a L con l'etichetta c. Questo processo viene ripetuto per un numero fisso di iterazioni o fino a quando non ci sono più dati da etichettare.". [12]

Nel pseudo-codice 1 è esplicitato quanto detto sopra:

Algorithm 1 self-training

```
repeat
    m ← train_model(L)
    for x ∈ U do
        if max m(x) > τ then
            L ← L ∪ {(x, p(x))}
until no more predictions are confident
```

1.5 Apprendimento con rinforzo

Il quarto e ultimo approccio è chiamato apprendimento con rinforzo, o reinforcement learning (RL), e si differenzia da quelli visti fino ad ora. Questo paradigma si occupa di problemi di decisione sequenziali, in cui l'azione da compiere dipende dallo stato attuale del sistema e ne determina quello futuro. In altre parole, questo è un sistema dinamico che può apprendere in seguito a ogni decisione presa, a prescindere che questa sia giusta o sbagliata.

Quando il sistema prende una decisione otterrà successivamente una "ricompensa" sotto forma di punteggio che sarà alto o basso a seconda che la decisione presa sia giusta o sbagliata. Con questa logica la macchina cercherà di fare sempre meglio per arrivare a ottenere il punteggio più alto possibile prendendo, così, solo le decisioni corrette.

1.6 Riduzione della dimensionalità

Nel ML, come abbiamo visto, l'input gioca un ruolo fondamentale nello sviluppo di un modello che riesca a predire nel modo corretto i nuovi dati che verranno esaminati da quest'ultimo. In questo lavoro ci concentreremo proprio sul riconoscimento dei volti all'interno delle immagini, quindi, per parlare dell'importanza del ridimensionamento delle *feature*, faremo riferimento proprio all'analisi di immagini che raffigurano persone, animali o oggetti.

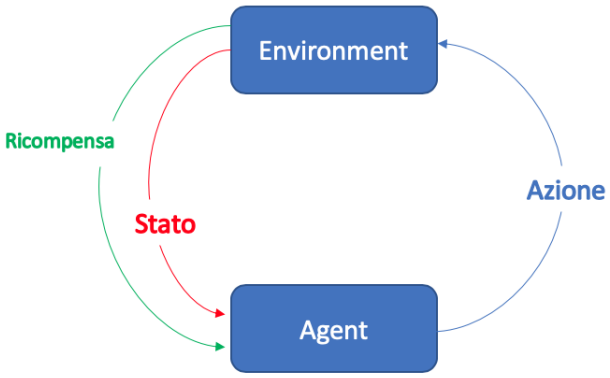


Figura 12: Scenario RL: l'Agent compie un'azione in un Environemnt che porta a un cambiamento di stato e a una ricompensa che influenzeranno le future scelte dell'Agent

Spesso ci troviamo di fronte a dati di dimensioni eccessive che comportano, *in primis*, dei problemi a livello tempistico e poi anche a livello computazionale. Ci basti pensare che quando cerchiamo di analizzare un'immagine per estrapolarne delle informazioni (riconoscimento di oggetti, persone, animali) dobbiamo passare in rassegna tutti i pixel! Supponiamo di prendere un'immagine a bassa risoluzione, ad esempio 500x500, ciò significherebbe trovarsi davanti a 500^2 pixel, ovvero 250.000 elementi per una singola immagine! Questo comporterebbe, quindi, analizzare uno spazio sovra dimensionato, con appunto 250.000 dimensioni.

Considerare uno spazio di quelle dimensioni è impensabile, proprio per questo ci vengono in soccorso delle tecniche che si occupano di ridurre il numero di elementi che definiscono l'oggetto estrapolandone solamente le informazioni più utili, informazioni che permettono di differenziare un oggetto da un'altro.

Pensiamo ad esempio a un'immagine in cui è raffigurato il volto di una persona. E' normale pensare che non tutti i pixel siano di fondamentale importanza per riconoscere il soggetto raffigurato. Ad esempio, tutti i pixel presenti nei bordi dell'immagine saranno sicuramente da scartare in quanto non ci diranno niente sulla persona raffigurata, così come molti altri pixel che raffigurano parti poco interessanti, come lo sfondo dell'immagine. Vediamo nei due paragrafi seguenti quali sono gli strumenti più utilizzati per risolvere questo tipo di problemi.

1.6.1 Analisi delle componenti principali

L'analisi delle componenti principali o *Principal Component Analysis* (PCA) è un metodo di riduzione della dimensionalità. Lo scopo di PCA è quello di diminuire

il numero di informazioni necessarie a descrivere un'osservazione limitando il più possibile la perdita di informazioni.

Consideriamo la matrice A in cui le colonne rappresentano le *feature* e le righe le osservazioni. In primo luogo, viene calcolata la media per ogni colonna della matrice A. Una volta calcolato il vettore risultante (la media) lo faremo coincidere con l'origine degli assi, in questo modo ogni punto verrà traslato di conseguenza. Successivamente calcoliamo la matrice di covarianza per A, la quale servirà per calcolare gli autovalori. Una volta ottenuto l'autovalore a questo punto viene calcolato l'autovettore il quale identifica la retta che meglio si adatta a tutti i punti, ovvero la retta R che minimizza la somma delle distanze dei punti da R. Questa viene chiamata PC_1. Si ripete questo passaggio per ogni dimensione, mantenendo la perpendicolarità della nuova retta (o PC_n) rispetto all'ultima retta calcolata (o PC_{n-1}). Una volta calcolate tutte le *principal component* dello spazio PCA, per ognuna di esse si utilizzerà l'autovettore per determinare la nuova posizione di ogni punto scalandolo sul rispettivo asse. Una volta scalati tutti i punti si calcola la varianza per ogni asse. Il valore della varianza σ^2 rispetto al i-esimo PC_i, calcolata in percentuale, ci dice quanto pesa l'informazione contenuta sulla dimensione i. Questo ci permetterà quindi, di eliminare gli assi meno interessanti, ovvero gli assi con la varianza in percentuale più bassa. La stessa informazione è di fatto contenuta negli autovalori calcolati precedentemente.

Questa tecnica, oltre a semplificare il lavoro di manipolazione delle *feature*, aiuta a migliorare i risultati degli algoritmo di ML poiché estrapola le informazioni realmente utili per predire la classe o il valore da attribuire ad un oggetto. Tutte le informazioni di contorno, come ad esempio i pixel situati sul bordo di un'immagine, possono essere fuorvianti per l'algoritmo di apprendimento. Questo è il motivo per cui la tecnica PCA semplifica e ottimizza i valori risultanti ed è ampiamente utilizzato nell'ambito di:

- riconoscimento facciale [2],
- image compression [3],
- rilevamento di pattern in campi ad alta dimensionalità [4],
- data mining [5].

1.6.2 t-Distributed stochastic neighbor embedding

t-Distributed stochastic neighbor embedding (t-SNE) è una tecnica di riduzione della dimensionalità non lineare che si presta particolarmente alla mappatura di spazi ad alta dimensionalità riducendoli in uno spazio a minori dimensioni. L'algoritmo modella i punti in modo che oggetti vicini nello spazio originale risultino vicini

nello spazio a dimensionalità ridotta e analogamente oggetti lontani nello spazio originale risultino lontani nello spazio a dimensionalità ridotta.

Per spiegare il funzionamento di questo algoritmo basiamoci su un caso semplice: un set bidimensionale S di dati. Con questo esempio spieghiamo quindi il funzionamento di t-SNE e come è possibile passare da due dimensioni ad una sola, trasformando coerentemente le distanze tra i punti osservati. Per farlo ci basiamo, inoltre, sulle probabilità che un elemento sia vicino ad un'altro. Per ogni punto x centriamo su di esso una curva gaussiana e inseriamo ogni altro punto $y \neq x$ sotto la distribuzione gaussiana centrata in x, per poi calcolarne la densità di probabilità (o *score*).

Viene utilizzata la curva gaussiana perché lavora bene su casi come questo: restituisce un valore elevato se un elemento è molto vicino a x e una valore molto basso se questo è lontano da x. A questo punto normalizziamo la curva per tutti i punti in modo che essi abbiano una misura proporzionata e non indipendente. La distribuzione può in realtà essere manipolata tramite una variabile denominata 'perplessità' (o *perplexity*), la quale modifica la varianza e quindi l'ampiezza della curva. In questo modo otteniamo la matrice quadrata M_1 con tutti gli *score* per ogni coppia. Ora inseriamo tutti i punti in maniera casuale su un'unico asse. Analogamente all'utilizzo che abbiamo fatto della curva gaussiana calcoliamo la probabilità di vicinanza tra i punti, con la differenza che questa volta useremo la distribuzione t di Student (in inglese Student t distribution, da cui deriva la t di t-SNE). Otteniamo così una seconda matrice quadrata M_2, la quale sarà, probabilmente, molto diversa da M_1. L'obiettivo adesso è quello di adattare la matrice M_2 a M_1 in modo che le distribuzioni di probabilità che descrivono i dati nello spazio ad alta dimensionalità e a bassa dimensionalità siano il più "vicini" possibili minimizzando la pseudo-distanza (anche detta "divergenza di Kullback-Leibler") tra queste. In questo modo siamo riusciti ad ottenere i cluster, visualizzabili nel grafico bidimensionale, in uno spazio monodimensionale.

Nella seconda parte dell'algoritmo, viene usata una t-distribution perché separa meglio i cluster nel piano generato. Se avessimo usato una distribuzione gaussiana, come nella prima parte, il risultato ottenuto sarebbe stato meno visibile, in quanto tutti i cluster si sarebbero ammassati al centro.

Questo è un esempio piuttosto semplice, ma nella realtà si possono adattare spazi a n dimensioni con $n \gg 2$ a spazi molto inferiori a n. Nel Capitolo 3 utilizzeremo t-SNE per passare da uno spazio di dimensioni elevate a uno spazio bidimensionale o al più a 5 dimensioni. Vedremo quindi, utilizzando questo algoritmo, quali sono le prestazioni che riesce a fornire nel caso del riconoscimento facciale e di classificazione dei soggetti.

Capitolo 2

Induzione di insiemi fuzzy

2.1 La logica fuzzy

La logica fuzzy (in italiano, logica sfocata), è un'estensione della logica booleana. Nella matematica booleana sono presenti solo due valori attribuili alle variabili: *vero* e *falso*. La logica fuzzy si definisce estensione della logica di Boole in quanto, al posto di prevedere solamente due possibili valori, viene previsto un insieme di valori continui compresi nell'intervallo [0, 1]. In questo intervallo, 0 corrisponde al valore *falso* e 1 al valore *vero*. Con questa estensione, oltre a poter dire *vero* o *falso* è possibile dire, tramite il grado di verità, quanto è vera una proprietà. Quindi, data una proprietà P e un elemento x, si può dire:

$$x \textit{ rispetta } P \textit{ con valore } y$$

dove y è compreso nell'intervallo $[0, 1]$.

Per fare un esempio più concreto si può pensare a tutte quelle cose che sono determinate in modo netto, in cui non esiste solo bianco o nero, bensì ci sono delle vie di mezzo più o meno vere. Supponiamo di prendere un oggetto di cui diciamo "essere freddo". Allora si può dire per esempio che:

- un gelato "è freddo" con valore (o grado di verità) uguale a 0.9,
- un bicchiere d'acqua a temperatura ambiente "è freddo" con valore uguale a 0.4,
- la resistenza di una lampadina accesa "è fredda" con valore uguale a 0.1.

2.2 Gli insiemi fuzzy

La logica fuzzy è strettamente legata alla matematica degli insiemi. Gli insiemi fuzzy sono un'estensione della teoria classica degli insiemi, secondo la quale un

elemento appartiene o meno ad un insieme. La teoria degli insiemi fuzzy è di fatto un'estensione della classica teoria degli insiemi con la differenza che questa, utilizza la logica fuzzy per valutare il valore di verità della proposizione "l'elemento x appartiene all'insieme A". Tale valore di verità prende il nome di grado di appartenenza dell'elemento all'insieme. Il grado di appartenenza degli insiemi fuzzy $x \in [0, 1]$ definisce quindi il grado di appartenenza ad un insieme. Per x pari a 1 l'elemento è, con un'alta probabilità, incluso nell'insieme, per x pari a 0 l'elemento è, al contrario, con una molta bassa probabilità, incluso nell'insieme, per tutti i valori compresi tra 0 e 1 l'appartenenza può essere più o meno forte.

Per fare un esempio definiamo lo spazio U come l'universo delle persone e un'insieme A che include tutte le persone giovani. Per ognuna delle seguenti categorie di persone in U:

- neonato
- ventenne
- ottantenne

si può definire per ognuna di esse un grado di appartenenza all'insieme A. Ad esempio:

- neonato appartiene ad A con un valore pari a 1
- ventenne appartiene ad A con un valore pari a 0.8
- ottantenne appartiene ad A con un valore pari a 0.1

Formalizzando quanto appena detto, definiamo

$$\mu_A : U \to [0, 1]$$

dove μ_A rappresenta una funzione, denominata funzione di appartenenza, che definisce il grado di appartenenza ad A per ogni elemento nell'universo dell'insieme U. Un insieme fuzzy è definito dalle coppie $(x, \mu_A(x))$, quindi dall'elemento x e il relativo grado di appartenenza ad A. Formalmente:

$$A = \{(x, \mu_A(x)) \mid x \in U\}.$$

2.3 Fuzzylearn

Nel capitolo precedente abbiamo visto le diverse tecniche utilizzate nel ML. L'algoritmo che andremo a descrivere, denominato *fuzzylearn*[6], si basa sull'induzione

di insiemi fuzzy e ricade nell'approccio supervisionato, ovvero quella tecnica che necessita di dati preventivamente valutati per effettuare predizioni.

In questo caso per predizione si intende il grado di appartenenza a un certo insieme fuzzy. Fissato un insieme fuzzy A, e dati due insiemi $X = \{x_1, ..., x_n\}$ di valori nell'universo di A e $M = \{\mu_1, ..., \mu_n\}$ dei corrispondenti gradi di appartenenza ad A, l'obiettivo di *fuzzylearn* consiste nel determinare un'approssimazione di A, e in particolare della sua funzione di appartenenza, partendo dai valori contenuti in X e M.

L'algoritmo *fuzzylearn* si basa sulla risoluzione di due problemi:

- determinare il sottoinsieme dell'universo in cui la funzione di appartenenza all'insieme fuzzy assume valore unitario,
- definire il comportamento della funzione di appartenenza nei casi rimanenti.

2.3.1 Calcolo dell'insieme fuzzy

Fissato un generico insieme fuzzy A, l'algoritmo *fuzzylearn* si basa sulle ipotesi descritte di seguito:

- Definiamo il core di A come l'insieme (classico) di tutti gli elementi dell'universo in corrispondenza dei quali il grado di appartenenza ad A è uguale a 1. Formalmente:

  $$c(A) = \{x \in U \mid \mu_A(x) = 1\}.$$

  Il core di A si può approssimare come l'insieme degli elementi di U le cui immagini tramite una funzione ϕ: $U \mapsto H$ sono incluse in una ipersfera.

- Indicato con a il centro dell'ipersfera descritta nel punto precedente, il grado di appartenenza $\mu_A(x)$ dipenderà solo dalla distanza di $\phi(x)$ da a.

Fatte queste ipotesi è possibile definire il seguente problema: trovare la più piccola ipersfera avente centro a e raggio R che include la maggior parte delle immagini tramite ϕ degli elementi $x \in X$. Viste le ipotesi precedenti, per tutti i punti x mappati in questa ipersfera varrà dunque $\mu_A(x) = 1$. La formulazione matematica di questo problema determina un problema di ottimizzazione vincolata, la cui funzione obiettivo è

$$\min R^2 + C \sum_{i=1}^{n} (\xi_i + \tau_i),$$

dove C è una costante di cui parleremo successivamente, mentre ξ e τ sono le variabili di scarto utilizzate nel problema di ottimizzazione. ξ è la variabile slack

legata al posizionamento dei punti all'interno dell'ipersfera mentre τ è riferita al posizionamento dei punti all'esterno dell'ipersfera.

Alla funzione obiettivo descritta sopra aggiungiamo i seguenti vincoli:

$$\mu_i \|\phi(x_i) - a\|^2 \leq \mu_i R^2 + \xi_i, \tag{1}$$

$$(1 - \mu_i)\|\phi(x_i) - a\|^2 \geq (1 - \mu_i)R^2 - \tau_i, \tag{2}$$

$$\xi_i \geq 0, \tau_i \geq 0, \tag{3}$$

dove possiamo notare che se il grado di appartenenza, vale a dire μ_i è pari a 1, otteniamo che in (1) si ha:

$$\|\phi(x_i) - a\|^2 \leq R^2 + \xi_i$$

ossia che la distanza dell'immagine di x_i da a ($\|\phi(x_i) - a\|^2$) è minore o uguale al raggio dell'ipersfera ($R^2 + \xi_i$) e che (2) diventa

$$\tau_i \geq 0,$$

la quale è già incorporata in (3) e quindi x_i è posizionato interamente a essa.

Al contrario, se μ_i è pari a 0 otteniamo che in (2) si ha:

$$\|\phi(x_i) - a\|^2 \geq R^2 - \tau_i$$

ovvero che la distanza di x_i da a è maggiore al raggio dell'ipersfera (a meno della i-esima variabile di scarto τ_i) e che (1) diventa

$$\xi_i \geq 0,$$

la quale, come per il caso di μ_i pari a 1, è già incorporata in (3). In generale, tramite questi vincoli, viene quindi determinata l'ipersfera in modo da tendere a includere i punti con alto grado di appartenenza, escludendo viceversa quelli con un grado basso.

Consideriamo il duale del problema sopra citato [10]. Formalmente otteniamo:

$$\max \sum_{i=1}^{n} (\alpha_i \mu_i - \beta_i(1 - \mu_i))k(x_i, x_i) - \sum_{i,j=1}^{n} (\alpha_i \mu_i - \beta_i(1 - \mu_i))(\alpha_j \mu_j - \beta_j(1 - \mu_j))k(x_i, x_j), \tag{4}$$

$$\sum_{i=1}^{n} (\alpha_i \mu_i - \beta_i(1 - \mu_i)) = 1, \tag{5}$$

$$0 \leq \alpha_i, \beta_i \leq C. \tag{6}$$

Indicando con α^* e β^* i valori ottimali delle variabili nel duale, si può dimostrare che il quadrato della distanza tra il centro dell'ipersfera ottimale e un generico punto $x \in X$ è pari a

$$R^2(x) = k(x,x) - 2\sum_{i=1}^{n}(\alpha_i^*\mu_i - \beta_i^*(1-\mu_i))k(x,x_i) + \sum_{i,j=1}^{n}(\alpha_i^*\mu_i - \beta_i^*(1-\mu_i))(\alpha_j^*\mu_j - \beta_j^*(1-\mu_j))k(x_i,x_j), \tag{7}$$

dove per ogni $a, b \in X$ si ha $k(a,b) = \phi(a) \cdot \phi(b)$. Con k si identifica la funzione kernel la quale permette di considerare una famiglia di possibili mapping ϕ. Di questa famiglia ne esistono diversi, ad esempio:

- lineare,
- polinomiale,
- gaussiano.

Il kernel lineare è uno dei più semplici e consiste nella moltiplicazione dei due vettori sommata ad una costante:

$$k(x,y) = x^T y + c.$$

Il kernel polinomiale invece, viene utilizzato per gestire problemi di suddivisone non lineare e si definisce come segue:

$$k(x,y) = (x^T y + c)^d,$$

con $c \in R$ e $d \geq 1$ dove d rappresenta il grado del polinimio e c una costante. Infine il kernel gaussiano, è un kernel usato nelle SVM ed è rappresentato da

$$k(x,y) = \exp\left(-\frac{\|x-y\|^2}{2\sigma^2}\right),$$

ed è anche il kernel che è stato utilizzato durante gli esperimenti di questo studio. Analogamente al Support Vector Clustering [9] in *fuzzylearn* si individuano i *support vector*, che in questo caso sono le immagini di x che giacciono sulla superficie dell'ipersfera. Inoltre, per tutti i punti x aventi $\mu(x)$ pari a 1 vale che la distanza di x da a (nella soluzione ottimale) è minore o uguale alla distanza di x_i

da a per tutti i *support vector* x_i. Formalmente possiamo riscrivere quanto appena detto come:

$$R^2(x) \leq R^2(s) \forall s \in S,$$

dove S definisce l'insieme di tutti i *support vector*. In generale, sulla base delle ipotesi fatte precedentemente, viene approssimato il core di A con tutte le pre-immagini dell'ipersfera che risolve il problema sopra definito.

Tutto quello che succede per i punti esterni all'ipersfera dipende da funzioni chiamate *fuzzifier*. Queste funzioni si assume che siano monotone non crescenti, le quali dipendono da un argomento, ovvero la distanza tra l'immagine di x e la frontiera dell'ipersfera. Quindi, in generale, con esse si descrive come si comporta $\mu(x)$ al crescere della distanza di x dall'ipersfera. Queste funzioni, come la costante C e il kernel k definiti precedentemente, vengono definiti iperparametri e il loro valore deve essere deciso prima di poter eseguire l'algoritmo *fuzzylearn*.

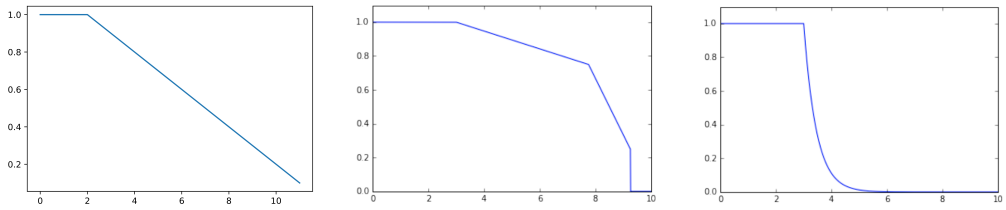
2.3.2 Configurazione degli iperparametri

Il termine "iperparametro" è ricorrente quando si parla di ML. Si definisce iperparametro un dato fornito dall'esterno all'algoritmo di ML (ML). In altre parole, tutti i dati forniti dall'utente che influiscono sul design del modello sono considerati iperparametri. Questi si differenziano dai cosiddetti parametri perché questi ultimi sono valori generati automaticamente dall'algoritmo stesso.

Abbiamo concluso il paragrafo precedente parlando di *fuzzifier*. Il *fuzzifier* è proprio un iperparametro. Fissato un insieme fuzzy A, esso definisce come si comporta la funzione di appartenenza per tutte le x che hanno $\mu_A(x) < 1$. Tramite questa funzione, quindi, siamo in grado di dire come decresce il grado di appartenenza di x ad A. Di seguito riportiamo degli esempi di *fuzzifier* [6] utilizzati in questo esperimento.

- Linear fuzzifier ($\hat{\mu}_{\text{lin}}$) è rappresentata da una funzione linearmente decrescente che decresce da 1 a 0 tanto velocemente quanto è la distanza del punto più distante dalla superficie della sfera (vedi Figura 13(a)).
- Quantile linear piecewise fuzzifier ($\hat{\mu}_{\text{qlin}}$) rappresenta la linearizzazione della suddivisione in quantili dei punti esterni alla sfera aventi come grado di appartenenza 1, 0.75, 0.5, 0.25, 0 rispettivamente (vedi Figura 13(b)).
- Exponential fuzzifier ($\hat{\mu}_{\text{exp}}$) è una funzione che decresce in modo esponenziale approssimando a 0 partendo dalla superficie dell'ipersfera (vedi Figura 13(c)).

Un secondo iperparametro che troviamo in *fuzzylearn* è la costante C che compare nella funzione obiettivo:



(a) $\hat{\mu}_{\text{lin}}$ (b) $\hat{\mu}_{\text{qlin}}$ (c) $\hat{\mu}_{\text{exp}}$

Figura 13: Esempio di *fuzzifiers* per un insieme I. L'asse delle ordinate rappresenta il grado di appartenenza e l'asse delle ascisse rappresenta la distanza dalla sfera. Al crescere della distanza diminuisce il grado di appartenenza all'insieme I secondo il *fuzzifier*.

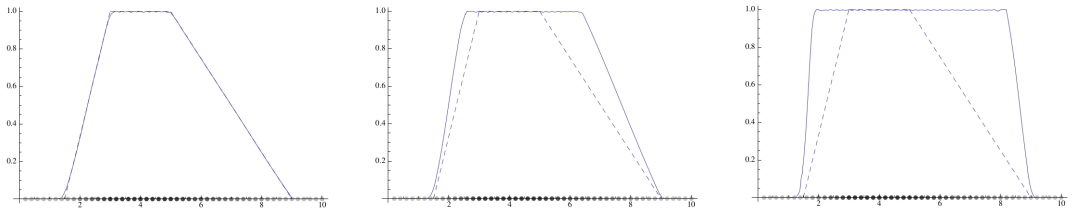
$$\min R^2 + C \sum_{i=1}^{n} (\xi_i + \tau_i)$$

Questa costante rappresenta un iperparametro che definisce il costo dell'errore nella classificazione degli elementi durante la costruzione dell'ipersfera. Questa C funziona in maniera analoga alla C nelle SVM (Support Vector Machine) [9]. Tanto più è grande C, tanto più sarà elevato il costo dell'errore nel posizionamento delle immagini dei punti dentro o fuori dall'ipersfera. Al contrario, tanto più è piccola C, tanto più piccolo sarà il costo dell'errore.

In Figura 14, possiamo osservare il ruolo di questo iperparametro al variare del suo valore. La spezzata tratteggiata è una funzione di appartenenza fissata, e sull'asse delle ascisse ci sono i valori di x, colorati in funzione del valore associato da questa funzione. La curva blu è invece la funzione di appartenenza appresa a partire dai punti, per valori diversi dell'iperparametro C. L'errore è quindi legato all'area tra la spezzata e la curva. Si vede come esista un valore di C che ci permette di ottenere un errore bassissimo (nel grafico a sinistra le due funzioni praticamente sono sovrapposte), e che all'aumentare di C, la funzione tende a diventare binaria e a identificare i dati più significativi per l'insieme (quelli per cui il grado di appartenenza non è nulla).

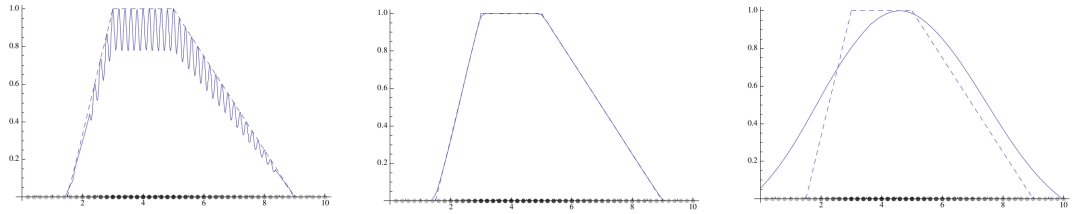
Esiste un ulteriore iperparametro che modifica la funzione di appartenenza, questo è il kernel. Questo iperparametro permette di modificare la forma di $\hat{\mu}$. Prendendo l'esempio in Figura 15 e una funzione kernel gaussiana, vediamo che effetto ha sulla forma della funzione di appartenenza la modifica del parametro σ del kernel.

Ricapitolando, l'algoritmo *fuzzylearn* prevede i tre seguenti iperparametri:



(a) $C = 0.0421$ (b) $C = 0.08$ (c) $C = 0.15$

Figura 14: Nelle figure è mostrato come incrementando C aumenti la sua "rigidità" aumenta la sua larghezza assomigliando sempre più ad una "funzione di appartenenza" di un insieme standard.



(a) $\sigma = 0.075$ (b) $\sigma = 0.12$ (c) $\sigma = 2$

Figura 15: Nelle figure è mostrato come incrementando σ si modifica la forma della funzione di appartenenza.

- C: per determinare il costo degli errori di classificazione in fase di definizione del fuzzy set,
- *fuzzifier*: funzione che determina il grado di appartenenza all'insieme fuzzy,
- kernel: funzione utile a mappare gli elementi dallo spazio originale ad uno spazio sovra-dimensionato.

Capitolo 3

Esperimenti

In questo capitolo ci concentremo sui risultati ottenuti tramite l'utilizzo di *fuzzylearn* su due diversi dataset di volti: Olivetti faces e AT&T faces. Descriveremo inoltre tutti i risultati commentandoli e traendo infine delle conclusioni sull'algoritmo utilizzato con le tecniche di riduzione PCA e t-SNE.

3.1 I dataset

3.1.1 Olivetti dataset

Olivetti dataset è un dataset di immagini di volti umani. Le immagini sono foto scattate presso AT&T Laboratories Cambridge tra il 1992 e 1994. Questo dataset è stato messo a disposizione proprio da AT&T Laboratories Cambridge. Nel dataset sono presenti 10 immagini per 40 soggetti distinti, ognuna delle quali raffigura quest'ultimo in diverse condizioni di luce, con differenti espressioni facciali (occhi chiusi/aperti, sorriso, ecc.) per un totale di 400 immagini. Ogni immagine è stata quantizzata a 256 livelli di grigio e con una risoluzione pari a 64 x 64. L'immagine è rappresentata da una lista di valori float (compresi tra 0 e 1) di 4096 elementi. Ogni soggetto è rappresentato tramite un numero che va da 0 a 39 che rappresenta l'etichetta di ogni record.

[FIGURA DI HEAD DEL DATASET]

3.1.2 AT&T faces dataset

Il dataset AT&T faces, come quello Olivetti, contiene anch'esso immagini raffiguranti 40 soggetti con 10 immagini distinte per ciascuno. Questo dataset, a differenza di Olivetti contiene immagini che sono grandi più del doppio di quelle raffigurate all'iterno del dataset di Olivetti. Ogni immagine a una risoluzione pari a 92 x 112, quindi con 10.304 pixel. Ogni immagine ha 256 livelli di grigio e ritrae

soggetti in diverse condizioni, esattamente come Olivetti. Ogni immagine è etichettata da un numero compreso tra 0 e 39 il quale rappresenta il soggetto.[nota: si ringrazia AT&T Laboratories per i dati messi a disposzione]

[FIGURA DI HEAD DEL DATASET?]

3.2 Valutazione del modello

Fino ad ora abbiamo visto come si costruiscono i modelli di ML e come possono essere configurati. In questo capitolo ci soffermeremo sulla valutazione di un modello e i relativi iperparametri configurati.

3.2.1 Training, Validation e Test set

Per costruire un modello abbiamo visto che è necessario un dataset su cui basare l'apprendimento. Il dataset che viene usato per la costruzione viene spesso diviso in due sotto-set: *training set* e *test set*.

Il *training set* è il sotto-set di dati utilizzata per allenare il modello. Quindi il modello vede e impara da questo set.

Il *test set* consiste nel set di dati del dataset escluso dal *training set*. Questo set, è utilizzato per valutare l'effettiva accuratezza del modello. Il *test set* quindi, viene usato per determinare lo *score* del modello nella generalità dei casi.

A volte si definisce un terzo sotto-set denominato *validation set*. È un set utilizzato per valutare le prestazioni con una specifica configurazione di iperparametri. Viene utilizzato quindi per una ricerca approfondita della migliore configurazione di questi. Una volta determinata la migliore configurazione del modello, questo viene nuovamente testato nel *test set* per vedere come si comporta effettivamente dopo una più approfondita ricerca dei migliori iperparametri.

3.2.2 Cross-validation

La *cross-validation* è un metodo statistico di valutazione delle performance. Con la *cross-validation* i dati dell'intero dataset vengono partizionati in k gruppi ognuno dei quali è denominato *fold* e dove k rappresenta un numero specificato dall'utente. Quando viene utilizzata questa strategia con un k pari a 5, significa che il dataset sarà diviso in 5 combinazioni differenti in 5 *fold*. Per ogni combinazione possibile un *fold* viene utilizzato come *test set* e gli altri come *training set*. Quindi il modello viene allenato per ogni combinazione data da questa strategia, utilizzando i $k-1$ fold per come *training set* e il restante *fold* come *test set*. In Figura è mostrato uno schema di una *cross-validation* a 5 fold.

[FIGURA DEL LIBRO DELLA CROSS VALIDATION 5.1]

L'utilizzo di questa strategia porta diversi vantaggi. Un primo importante vantaggio è rappresentato dal fatto che l'accuratezza del modello è calcolata sulla media di tutti i valori restituiti da ogni *test set* nelle varie combinazioni dei *fold*. Per capire meglio questo vantaggio, vediamo l'esempio seguente. Supponiamo di essere "fortunati" e quindi di allenare il modello su un *training set* che contiene solo dati di difficile classificazione. Questo significa che il *test set* probabilmente avrà solamente dei dati molto facili da classificare il che permette al modello di avere prestazioni (non veritiere) molto alte. Al contrario se siamo "sfortunati" avremo che tutti i dati di difficile classificazione sono all'interno del *test set* e quelli semplici nel *training set*. In questo modo avremo risultati piuttosto bassi. Con l'utilizzo di *cross-validation* però, la possibilità di avere casi come questi sarà molto bassa proprio perchè verranno usati dati di allenamento sempre diversi per uno stesso modello. Il risultato finale sarà quindi composto dalla media di tutti i risultati delle varie combinazioni.

Un'altro vantaggio corrisponde al fatto che oltre a variare l'utilizzo dei dati di allenamento, varia anche la dimensione di questi. Se utilizziamo 10 *fold* ad esempio, significa che il *test set* sarà definito da un decimo della dimensione totale del dataset. Analogamente se ne utilizziamo 5, il *test set* sarò costituito da un quinto del totale.

La *cross-validation* presenta però un grosso svantaggio nelle performance. Questo perchè maggiore sarà il numero di *fold* e maggiore sarà il tempo richiesto per ottenere un risultato. In altre parole con k *fold* avremo un tempo pari a k volte il tempo necessario ad allenarlo.

3.2.3 Grid Search

Ora che abbiamo capito come avviene e su cosa si basa una buona valutazione del modello possiamo passare al prossimo step: la ricerca del miglior modello. Cercare il miglior modello significa cercare la migliore configurazione di iperparametri che meglio generalizzano il problema originale. *Grid Search* è una tecnica molto usata per la ricerca del miglior iperparametro. Questa tecnica consiste nel provare ogni possibile combinazione di iperparametri di nostro interesse. Consideriamo il caso di *fuzzylearn*. Come abbiamo visto nel capitolo precedente vi sono tre differenti iperparametri che caratterizzano l'algoritmo: *C*, *kernel function*, *fuzzifier*. Supponiamo di voler testare i valori 0.001, 0.1, 1, 1000 per C, un kernel gaussian con sigma pari a .225, .5, 1, 2 per il kernel e Linear fuzzifier, CrispFuzzifier per il fuzzifier. Così facendo abbiamo un totale di $4x4x2 = 32$ combinazioni differenti di iperparametri. Iterano tutte le possibili combinazioni su un *training set* e valutandone le prestazioni su un *test set* possiamo quindi trovare quale fra tutte le 32 combinazioni restituisca il valore più alto.

Questa tecnica ha degli svantaggi perchè i risultati ottenuti non sempre sono del tutto veri. Questo succede perchè nel testare le prestazioni delle diverse configurazioni consideriamo *test set* differenti per ognuna di queste. Così facendo i risultati ottenuti si basano tutti sul prorpio test set e questo non ci permette di avere un'unica misura per valutrare le effettive prestazioni.

Per risolvere questo problema si ricorre all'utilizzo del *validation set* che permette quindi di determinare quale possa essere la migliore configurazione e successivamente testarne le effettive prestazioni su dati che non sono stati usati per aggiustare la configurazione. In Figura zzz è mostrata la suddivisione del dataset con l'aggiunta del *validation set*.

[FIGURA DI DATASET IN TRAIN TEST E VALIDATION 5.5]

3.2.4 Grid Search con Cross-Validation e Nested Grid Search

Nei due paragrafi precedenti abbiamo visto quale sia una buona valutazione delle performance del modello e successivamente di come si cerchi efffettivamente una buona configurazione di iperparametri. Queste due tecniche vengono spesso utilizzate insieme per vedere in generale quale sia la migliore combinazione possibile per un modello utilizzando diversi dati di *training* e di *test*. Quindi quello che viene fatto è, per ogni modello, provare su ogni combinazione di dati per il training e il test come funzione ogni combinazione di iperparametri. Questo consente, quindi, un ulteriore grado di valutazione del modello. Il grosso svantaggio di combinare queste due tecniche consiste nel tempo richiesto per la computazione. Per possibilearn C, sigma, fuzzifier come iperparametri configurabili. Supponendo di provare per C i seguenti valori: [LISTA DI IPERPARAMETRI DI C] , per sigma i valori: [LISTA DI IPERPARAMETRI DI SIGMA] e mantendo il *fuzzifier* fisso, otteniamo 5 x 5 = 25 possibili combinazioni mostrate in Figura xxx. [TABELLA DI TUTTI LE COMBINAZIONI DEGLI IPERPARAMETRI] A questo le 25 configurazioni dobbiamo moltiplicalre per il numero di fold che usiamo nella cross validation. Prendendo, ad esempio, 5 fold otteniamo 25 x 5 = 125 diversi modelli da costruire! È importante quindi utilizzare il Grid Search unito alla Cross-Validation prestando attenzione a quali iperparametri si vogliono confrontare per evitare di non fare computazioni eccessivamente lunghe.

3.3 I risultati

In conclusione analizziamo i risultati ottenuti sui dataset Olivetti e AT&T tramite *fuzzylearn*. Nella ricerca e nello studio di come si comporta *fuzzylearn* nel riconoscimento facciale e di classificazione dei soggetti sono stati usati tre computer per ridurre i costi temporali richiesti dall'allenamento di ogni modello.

Inizialmente è stata costruita un'infrastruttura client-server il cui server godeva di alte performance computazionali. Purtoppo queste non sono state sufficienti in quanto i tempi di calcolo erano troppo lunghi. Quindi la computazione è stata in seguito delegata a tecnologie di Google denominate Google Colaboratory le quali hanno di una maggiore performance computazionale permettendo il risparmio di risorse temporali.

Per il calcolo dei risultati, a livello pratico, è stato usato *python 3.7* come lignuaggio di programmazione il quale rappresenta uno dei più utilizzati linguaggi nell'ambito del ML per la sua comodità e semplicità di utilizzo e perchè sono presenti molte librerie ricche di funzionalità sviluppate proprio in *python*. I pacchetti usati durante questi studi sono stati *scitkit_learn* per l'elaborazione dei dati e *matplotlib* insieme a *plotly* per la produzione di grafici 2D e 3D. In particolare sono state usate le seguenti librerie/funzionalità:

- *GridSearchCV*: funzione di *scikit_learn* che, presa una lista di valori per ogni iperparametro, applica su diverse suddivisioni del dataset la combinazione di ognuna di queste per trovare poi quella che definisce il modello con lo score medio più alto (come spiegate in **??**),
- *pre-processing*: libreria di *scikit_learn* utile ad applicare delle trasformazioni ai dati prima che questi vengano usati per allenare il modello. In particolare *StandardScaler* è una funzione di questa libreria per standardizzare le *feature* e *MinMaxScaler* per adattare a uno specifico intervallo (comune a tutte le *feature*) i singoli valori delle *feature*,
- *PCA*: funzione di *scikit_learn* che, come si può intendere dal nome, serve ad applicare una riduzione di dimensionalità tramite PCA
- *TSNE*: funzione di *scikit_learn* che, come si può intendere dal nome, serve ad applicare una riduzione di dimensionalità tramite TSNE
- *pyplot*: libreria di *matplotlib* che ha avuto un utilizzo speicifico nella produzione di grafici 2D quali grafici di dispersione (o *scatter plot*), grafici di contorno e grafici a linea,
- *express*: libreria di *plotly* per la produzione di grafici tridimensionali come grafici di dispersione 3D e grafici di contorno 3D

Inoltre è stato usato il pacchetto di *fuzzylearn* (disponibile su github all'indirizzo https://github.com/dariomalchiodi/fuzzylearn) il quale contiene il codice per l'utilizzo di *fuzzylearn*. Il modello codificato in questo pacchetto è stato adattato alle librerie di *scikit_learn* per poterne utilizzare le funzionalità di queste. Nello specifico l'adattamento richiede l'impkementazione delle seguenti funzionalità:

- *fit(X, y)*: la funzionalità principale di tutti i modelli di ML. *fit* è utilizzata per allenare un modello sui dati X conoscendone le rispettive etichette y,
- *predict(X)*: contiene l'implementazione per predire la classificazione di ogni elemento di X, dove X è rappresentato da un vettore di elementi. Nel caso specifico di *fuzzylearn*, *predict* restituisce, per ogni x di X, la probabilità (tra 0 e 1) che x appartenga al modello,
- *score(X, y)*: contiene l'implementazione dell'accuratezza di predizione del vettore X sapendo che le relative etichette sono y. In particolare *fuzzylearn* restituisce uno score pari alla differenza al quadrato della probabilità predetta dal modello e l'effettiva probabilità di appartenenza

Nei prossimi due paragrafi vediamo i risultati ottenuti tramite l'utilizzo delle diverse tecniche di riduzione di dimensionalità PCA e t-SNE. Per ogni tecnica vediamo i risultati ottenuti sia per Olivetti dataset che per AT&T dataset. Inoltre, durante la descrizione dei risultati vediamo anche l'uso di alcuni degli strumenti sopra citati.

3.3.1 Utilizzo metodo di riduzione t-SNE

In questo paragrafo ci concentriamo sui risultati otteuti applicando una riduzione tramite TSNE a 2 e 3 dimensioni. La scelta di queste dimensioni è dovuta al fatto che t-SNE permette di trasformare uno spazio a n dimensioni in uno spazio più piccolo mantenendo (e quindi focalizzandosi) le distanze tra i punti dello spazio originale. Per questo motivo è stato ritenuto più interessante ridurre le dimensioni originali in due dimensioni proiettabili in uno spazio naturale per farne poi un'analisi a livello qualitativo più immediata. Vediamo quindi la distribuzione dei punti quando vengono ridotti in due dimensioni.

[SCATTER PLOT 2D OLIVETTI E AT&T]

In Figura [SCATTER PLOT 2D OLIVETTI E AT&T] sono mostrati i grafici di dispersione per entrambi i dataset dopo aver applicato t-SNE a 2 dimensioni. Analizzando quindi i grafici è piuttosto immediata la distinzione dei vari gruppi rappresentanti ogni soggetto. Partendo da una considerazione del tutto qualitativa e visiva possiamo aspettarci come la costruzione dei modelli, con i giusti iperparametri, ci possa restituire una buona accuratezza nella classificazione dei soggetti.

[SCATTER PLOT 3D OLIVETTI E AT&T]

Diverso è, il caso riportato in Figura [SCATTER PLOT 3D OLIVETTI E AT&T] in cui sono mostrati i grafici di dispersione in 3 dimensioni. È facile notare come i cluster siano meno distinguibili rispetto alla proiezione in due dimensioni e quindi è naturale pensare che i risultati di classificazioni su 3 dimensioni siano

inferiori rispetto a quelli su dati in 2 dimensioni. In Tabella [TABELLA TSNE OLIVETTI E AT&T] sono mostrati i rispettivi risultati: le colonne indicano i set su cui è stato testato l'insieme dei modelli mentre le righe indicano la dimensioni di riduzione raggiunte.

[TABELLA TSNE OLIVETTI E AT&T]

Con la tabella è stato confermato quanto previsto nelle considerazioni visive fatte sui grafici di dispersione in Figura [SCATTER PLOT 2D OLIVETTI E AT&T] e in Figura [SCATTER PLOT 3D OLIVETTI E AT&T]. I risultati ottenuti sono frutto di 10 holdout ripetuti con una dimensione per il *test set* pari al 20% del totale, di conseguenza il restante 80% è dedicato al *training set*.

3.3.2 Utilizzo metodo di riduzione PCA

Ora vediamo come si comporta *fuzzylearn* con i dati ridotti tramite PCA. In questo caso abbiamo considerato un numero maggiore di dimensioni rispetto a t-SNE perchè la logica che sta dietro a questo algoritmo è differente da t-SNE. Come abbiamo già visto nel Capitolo 1, PCA riduce le dimensioni eliminando le caratteristiche che contengono la minor quantità di informazioni. In Figura [PLOT LINE PCA RATIO] vediamo i grafici che mostrano come, la quantità di informazione, decresce al crescere di dimensioni da poter utilizzare. La scelta del numero di dimensioni è partita da un'analisi fatta su questo grafico il quale permette di capire fino a dove ha senso spingersi per cercare di ottenere il maggior numero di informazioni mantenendo la complessità del modello (e quindi il numero di caratteristiche che descrivono ogni osservazione) più bassa possibile. Con PCA sono state sperimentate le riduzioni a 2, 5, 20, e 50 dimensioni. L'unica riduzione che è possibile analizzare è, naturalmente, quella a due dimensioni. Pertanto in Figura [SCATTER PLOT PCA OLIVETTI E AT&T] è possibile vedere la distribuzione delle osservazioni nel piano. A differenza di t-SNE, in questi grafici si riesce a distinguere solo qualche piccolo cluster e, oltretutto, facendo fatica. Questo non è un buon segno, infatti come vediamo nei risultati più avanti l'accuratezza di classificazione data dall'insieme dei 40 modelli è decisamente di scarsa qualità. Uno dei motivi per cui la differenza è così marcata tra t-SNE e PCA sta proprio nel fatto che le due tecniche affrontano lo stesso problema, ovvero quello di ridurre le dimensioni, analizzando aspetti totalmente diversi. Mentre t-SNE confronta le distanze tra i punti in un iperspazio, PCA va a selezionare solo le *feature* che mantengono la maggior quantità di informazioni. Ritornando al grafico in Figura [PLOT LINE PCA RATIO] vediamo che utilizzando le prime due dimensioni otteniamo solamente il 30

[TABELLA PCA OLIVETTI E AT&T]

Nella tabella sopra riportata è infatti possibile notare un certo andamento dei risultati che è direttamente proporzionale al numero di feature utilizzate. Questo segue esattamente quanto scritto sopra: con l'aumentare delle dimensioni aumentano le caratteristiche che descrivono l'oggetto.

3.3.3 Analisi conclusiva

Nell Tabelle [TABELLA TSNE OLIVETTI E AT&T] e [TABELLA PCA OLIVETTI E AT&T] abbiamo visto quale sia l'accuratezza fornita da *fuzzylearn* sui dataset Olivetti e AT&T mediante le due tenciche di riduzione PCA e t-SNE. Da questi risultati si evince che entrambe le tecniche portano a buoni e interessanti risultati con la differenza che gli spazi da considerare sono molto diversi. Con t-SNE abbiamo potuto osservare la sua potenzialità nell'immediatezza visiva di rappresentazione dei dati, il quale è un ottimo strumento per poter fare analisi qualitative dei dati. Con PCA, invece, è stato possibile vedere come la "qualità", intesa come la quantità di informazioni contenute nei dati, influsica sull'accuratezza dei modelli allenati su di essi. Infine possiamo dire che *fuzzylearn* è un ottimo algoritmo che permette, nel caso specifico di riconoscimento facciale, buone prestazioni a detrimento del tempo richiesto per allenare i modelli. Gli aspetti positivi ricadono, quindi, nella possibilità di rendere i modelli molto "flessibili" grazie ai diversi iperparametri di cui gode e, quindi, di adattarsi bene a contesti complessi come il riconoscimento di volti in immagini, mentre lo svantaggio principale consiste nel tempo richiesto per l'allenamento del modello.

Bibliografia

[1] Andreas C. Müller, Muller Andreas C, Sarah Guido. *Introduction to Machine Learning with Python: A Guide for Data Scientists*. "O'Reilly Media, Inc.", 26 Settembre 2016

[2] Yagnesh Parmar. *3D Face Recognition Using PCA: The Robust Face Recognition System Using Matlab*. Lap Lambert Academic Publishing GmbH KG, 2012

[3] Fred Hohman. *Image Compression via PCA*, `https://fredhohman.com/assets/image_compression.pdf`

[4] Niels da Vitoria Lobo, Takis Kasparis, Michael Georgiopoulos, Fabio Roli, James Kwok, Georgios C. Anagnostopoulos, Marco Loog. *Structural, Syntactic, and Statistical Pattern Recognition*. Springer Science & Business Media, 24 nov 2008

[5] Djamel A. ZighedJan KomorowskiJan Żytkow. *Principles of Data Mining and Knowledge Discovery*. Springer, Berlin, Heidelberg

[6] Dario Malchiodi, Witold Pedrycz. *Learning Membership Functions for Fuzzy Sets through Modified Support Vector Clustering* F. Masulli, G. Pasi, and R. Yager (Eds.): WILF 2013, LNAI 8256, pp. 52–59, 2013. © Springer International Publishing Switzerland 2013

[7] Dario Malchiodi, Andrea G. B. Tettamanzi. *Predicting the Possibilistic core of OWL Axioms through Modified Support Vector Clustering* In SAC 2018: Symposium on Applied Comput-ing , April 9–13, 2018, Pau, France. ACM, New York, NY, USA,8 pages. https://doi.org/10.1145/3167132.31673451

[8] Rushikesh Pupale. *Support Vector Machines(SVM) — An Overview*, `https://towardsdatascience.com/https-medium-com-pupalerushikesh-svm`

[9] Ben-Hur A., Weston J. (2010) *A User's Guide to Support Vector Machines*. In: Carugo O., Eisenhaber F. (eds) Data Mining Techniques for the Life Sciences.

Methods in Molecular Biology (Methods and Protocols), vol 609. Humana Press

[10] R. Fletcher. *Practical Methods of Optimization*, Copyright © 1987 by John Wiley & Sons, Ltd

[11] Jake VanderPlas. *Python Data Science Handbook: Essential Tools for Working with Data*. "O'Reilly Media, Inc.", 6 dicembre 2016

[12] Sebastian Ruder. *An overview of proxy-label approaches for semi-supervised learning*
`https://ruder.io/semi-supervised/`

[13] Knuth: Computers and Typesetting,
`http://www-cs-faculty.stanford.edu/~uno/abcde.html`