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.

1.1.1 Supervisionato

L'approccio supervisionato 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 barca", tutte le predizioni (su questo dataset) saranno corrette.

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

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.

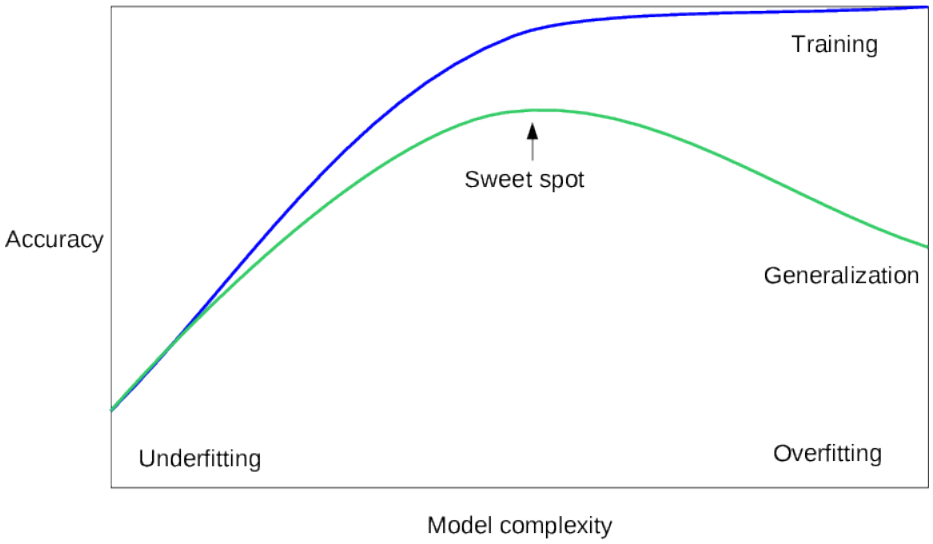


Figura 1: Trade-off tra overfitting e underfitting

1.1.1.1 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 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.

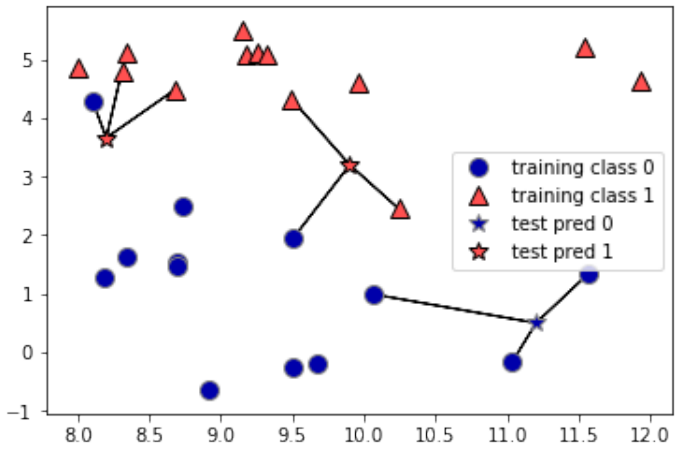


Figura 2: Classificazione tramite k-NN di 3 elementi. Le stelle rosse rappresentano una classificazione rispetto alla classe 1 (dei triangoli) mentre le stelle blu rappresentano una classificazione rispetto alla classe 0 (dei cerchi).

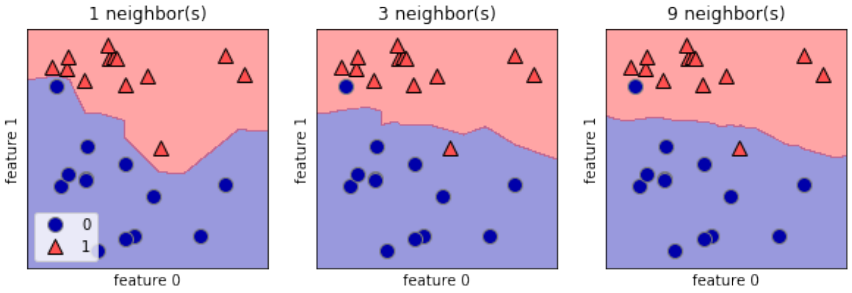


Figura 3: Risultati di un classificatore k-NN, per diversi valori del parametro k su un medesimo dataset. I cerchi e i triangoli indicano le osservazioni del dataset appartenenti a due classi, mentre le aree blu e rosse indicano gli esiti della classificazione.

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 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

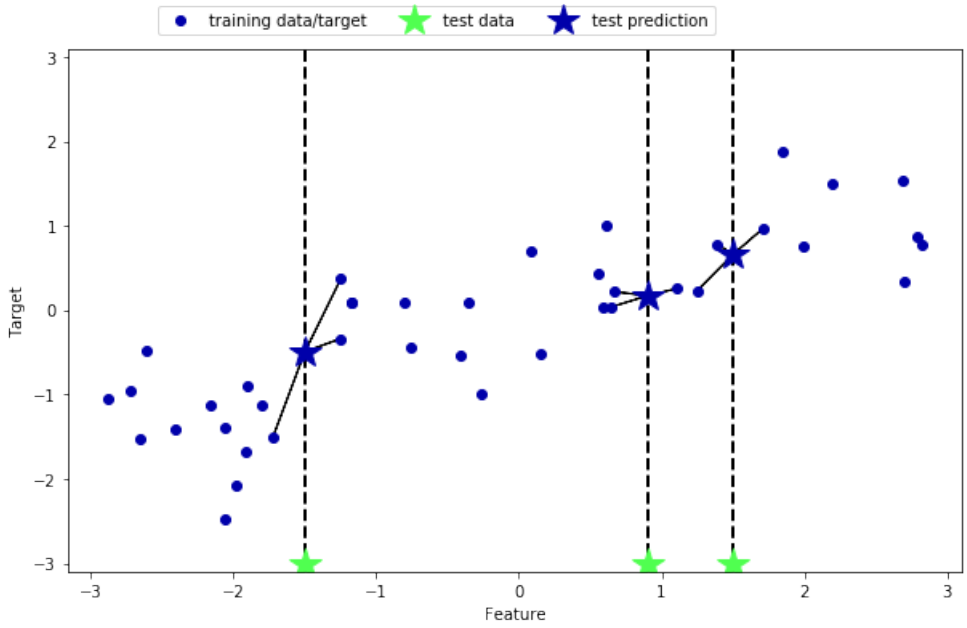


Figura 4: Risultato di un regressore k-NN, per k pari a 3. I cerchi indicano le osservazioni del dataset, le stelle verdi indicano il nuovo elemento da predire e le stelle blu gli esiti della predizione.

dei 3 elementi più vicini sull'asse delle ascisse (l'asse delle *feature*).

1.1.1.2 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:

$$y = w_0 x_0 + w_1 x_1 + ... + w_n x_n + b,$$

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:

$$y = w_0 x_0 + b,$$

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.

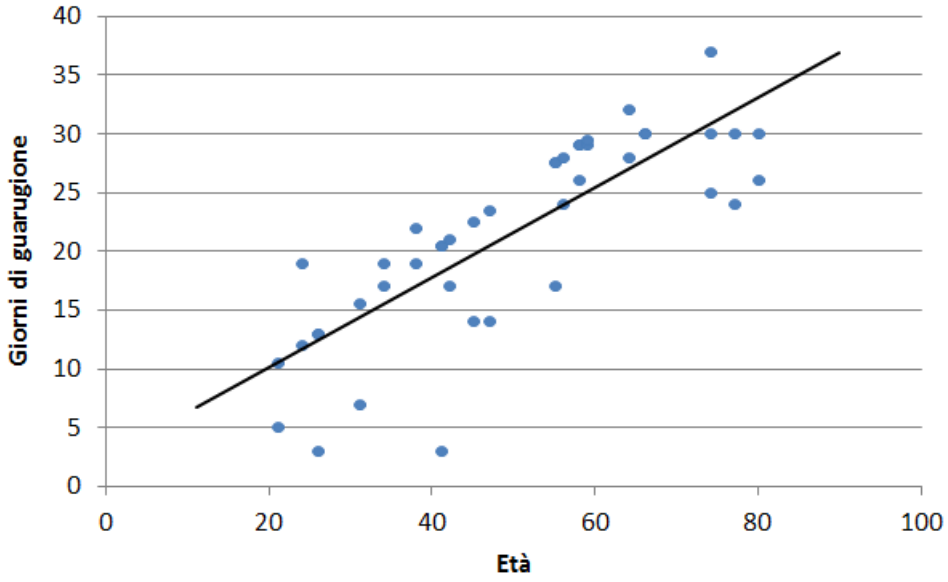


Figura 5: Regressione lineare. I punti rappresentano i pazienti e la retta nera rappresenta la retta di regressione che approssima meglio all'andamento di tutti i punti

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.

1.1.1.3 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.

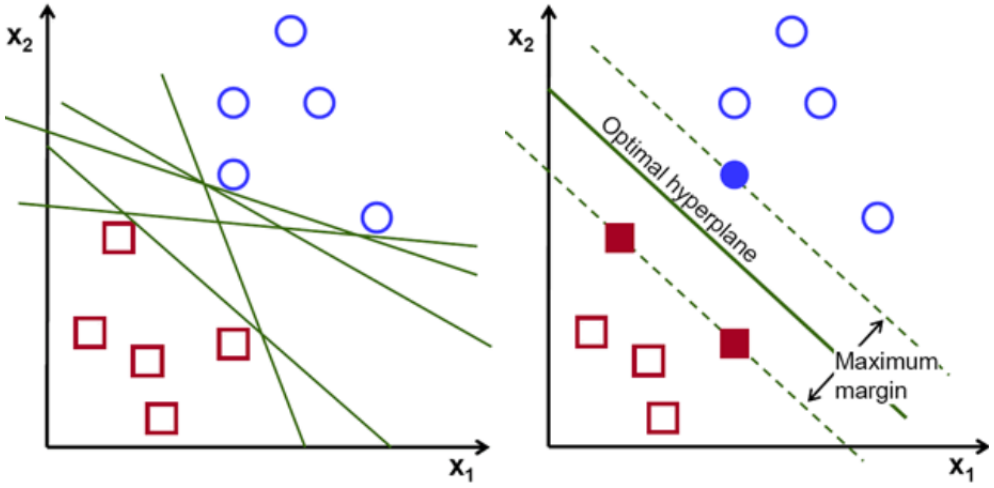


Figura 6: Possibili iperpiani che dividono lo spazio (a sinistra) e iperpiano "migliore" (a destra)

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 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:

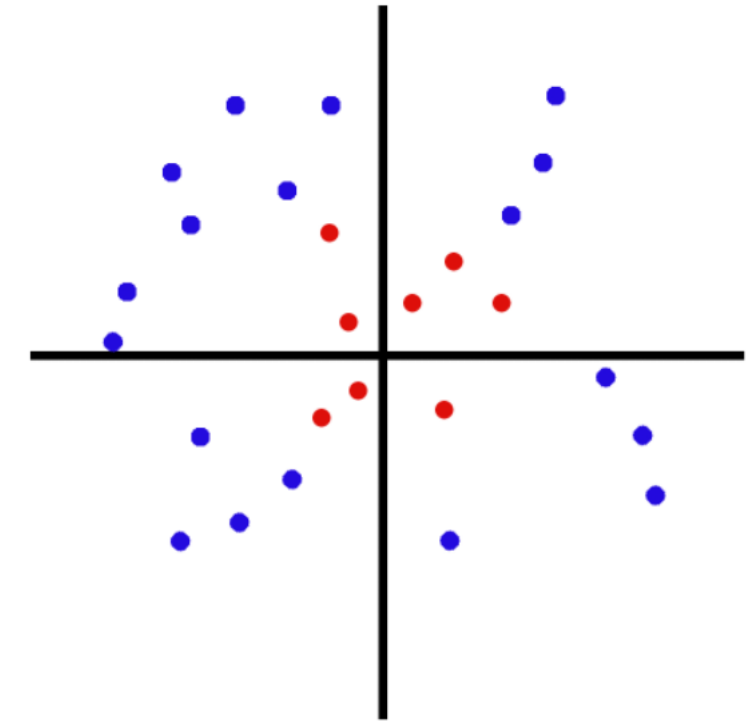


Figura 7: Dati non divisibili linearmente

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

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*.

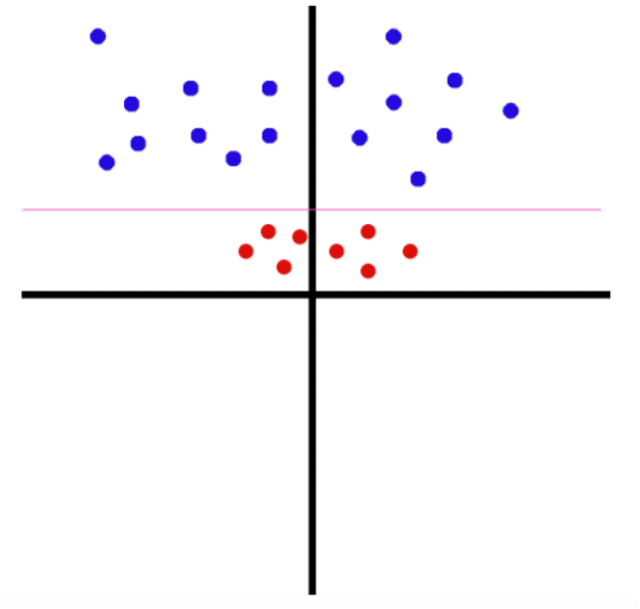


Figura 8: Dati visti sull'asse z

1.1.1.4 Alberi di decisione

Gli alberi di decisione (o *decision trees*) sono un algoritmo 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.

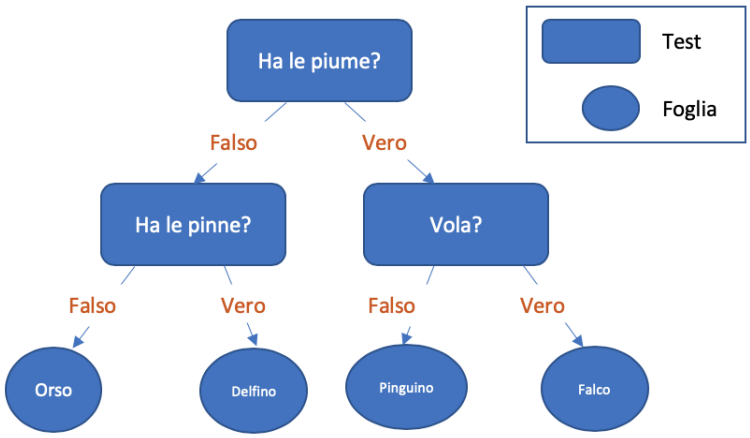


Figura 9: Esempio di albero di decisione per la classificazione di un animale

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"); 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 determinarne 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 è l'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* a cui rispondono 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 il piano su cui rappresentiamo i dati in diverse aree, ognuna delle quali è rappresentata da una foglia (e quindi da una classe).

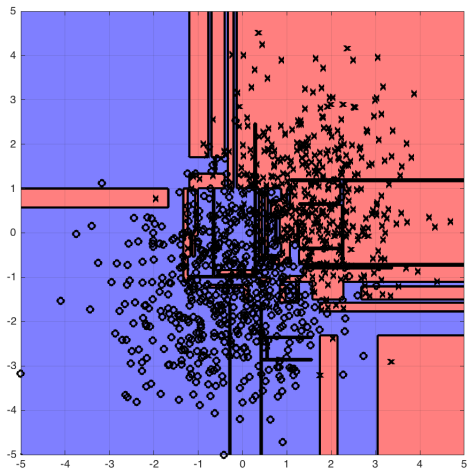


Figura 10: Overfitting dei dati con l'albero di decisione

Come abbiamo già visto negli altri algoritmi, il problema dell'*overfitting* e *underfitting* è un problema ricorrente che si presenta anche nel caso dell'albero 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 sottoforma di aree 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. Questo succede proprio perché il modello è stato allenato adattandosi in maniera eccessiva all'insieme di dati di allenamento.

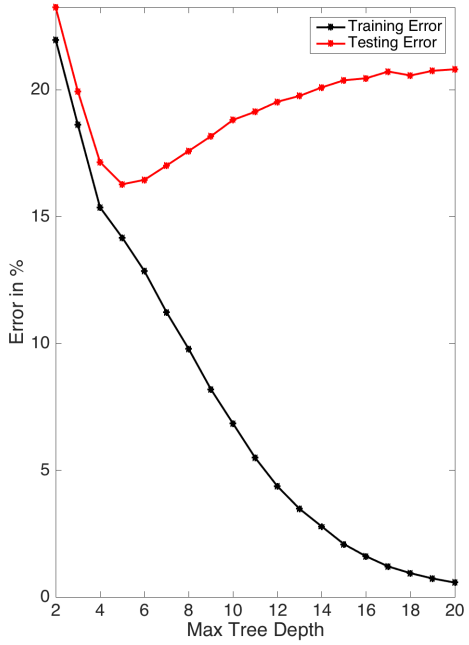


Figura 11: Grafico di overfitting con l'albero di decisione. La curva nera rappresenta l'errore di precisione nella predizione di dati di allenamento mentre la curva rossa rappresenta l'errore sui dati di test

Per risolvere questo problema, esistono due strategie:

- far terminare lo sviluppo dell'albero dopo pochi passi (pre-potatura) ovvero limitare la profondità dell'albero mediante una variabile che tenendo conto della profondità dello stesso ne interrompa lo sviluppo al superamento di una soglia prefissata,
- rimuovere i nodi che contengono informazioni poco significative (post-potatura).

1.1.2 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 un algoritmo 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 applica proprio un algoritmo di clustering. In che modo? L'algoritmo non sa nè chi siano le persone raffigurate nè 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 come questa possiamo riuscire ad individuare e bloccare attacchi tuttora sconosciuti. Supponiamo di essere loggati nel sito della nostra banca. Tramite il ML non supervisionato il sistema memorizzerà tutte le operazioni che faremo. Supponiamo di effettuare quotidianamente delle specifiche operazioni ad un fissato orario, in una determinata località geografica. Bene, queste informazioni vengono salvate dal calcolatore il quale le utilizzerà per creare dei cluster, ossia per riconoscerci sulla base delle nostre operazioni ricorrenti tramite orario, località geografica e altre possibili informazioni. Supponiamo ora che un malintenzionato, dall'altra parte del mondo, ad 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.

1.1.3 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 e gli utenti che sono in grado di etichettarli i dati sono 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 ad 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 definiti secondo un particolare modo e altri che sono privi di definizione. 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 [4] 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."

Di seguito vediamo uno pseduo-codice che esplicita quanto detto sopra:

1: **repeat**
2: $m \leftarrow train\_model(L)$
3: **for** $x \in U$ **do**
4: **if** $\max m(x) > \tau$ **then**
5: $L \leftarrow L \cup \{(x, p(x))\}$
6: **until** no more predictions are confident

1.1.4 Apprendimento con rinforzo

Il quarto ed ultimo approccio chiamato Apprendimento con rinforzo, o *Reinforcement Learning* (RL), è un approccio che 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 ad ogni decisione presa, a prescindere che questa sia giusta o sbagliata.

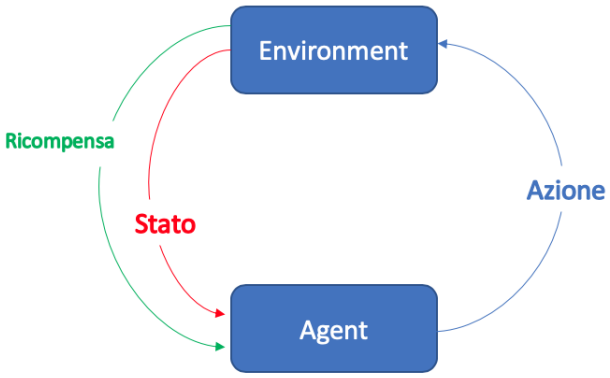


Figura 12: Scenario RL

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.

1.2 Ridimensionamento delle funzionalità

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.

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.

1.2.1 Principal Component Analysis

Principal Component Analysis (PCA) è un metodo di riduzione della dimensionalità. Lo scopo di PCA è quello di diminuire il numero di variabili limitando il più possibile la perdita di informazioni.

In primo luogo, viene calcolata la media per ogni *feature*. 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. A questo punto viene calcolata 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 si calcola, per ognuna di esse, l'autovettore, il quale sarà utilizzato 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 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.

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
- image compression
- rilevamento di pattern in campi ad alta dimensionalità
- data mining

1.2.2 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, mantenendo le corrette distanze. 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 tale che:

$$y \in S \setminus \{x\},$$

sotto la distribuzione gaussiana centrata in x, per poi calcolarne la probabilità di densità (o *score*).

Viene utilizzata la curva gaussiana perché lavora bene su casi come questo: restituisce un'alta probabilità se un elemento è molto vicino e una probabilità molto bassa se questo è lontano. 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. A questo punto 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 di Student (in inglese 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 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, ad 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 lo 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 valore di appartenenza (o 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 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. Secondo la logica degli insiemi fuzzy $x \in [0, 1]$ definisce il valore di appartenenza ad un insieme. Per x pari a 1 l'elemento è certamente incluso nell'insieme, per x pari a 0 l'elemento è sicuramente escluso dall'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 ed un'insieme A che include tutte le persone giovani. Prendendo le seguenti persone di 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 apparteiene ad A con un valore pari a 0.1

Formalizzando quanto appena detto, definiamo:

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

dove μ_A rappresenta la funzione di appartenenza ad A ed U rappresenta lo spazio considerato. 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 Possibilearn

Nel capitolo precedente abbiamo visto le diverse tecniche utilizzate nel ML. L'algoritmo che andremo a descrivere, denominato *possibilearn*, si basa sull'induzione di insiemi fuzzy e ricade nell'approccio supervisionato, ovvero quella tecnica che necessita di dati preventivamente classificati per effettuare predizioni.

In questo caso per classificazione si intende il grado di appartenenza ad un certo insieme fuzzy. Dato un insieme $X = \{x_1, x_2, \ldots, x_{n-1}, x_n\}$ e il relativo insieme di gradi di appartenenza $M = \{\mu_1, \mu_2, \ldots, \mu_{n-1}, \mu_n\}$, l'obiettivo di *possibilearn* consiste nel determinare la forma dell'insieme fuzzy tramite l'utilizzo di X e M.

Questo richiede di risolvere due problemi:

- calcolare l'insieme fuzzy,
- definire i parametri per la funzione di appartenenza all'insieme fuzzy.

2.3.1 Calcolo dell'insieme fuzzy

Partendo dalle ipotesi che:

- A è l'insieme fuzzy che contiene tutti gli elementi con grado di appartenenza pari a 1. Formalmente:

  $$A = \{x \in X \mid \mu_A(x) = 1\}$$

  dove X è l'insieme delle immagini dei punti dello spazio originale appartenenti una ipersfera di centro a e raggio R che sono mappate tramite una funzione ϕ.

- Il grado di appartenenza $\mu_A(x)$ dipenderà solo dalla distanza di $\phi(x)$ da α.

Fatte queste ipotesi è possibile definire il seguente problema: *trovare la più piccola ipersfera avente centro a e raggio R che include tutte le x in X, per cui vale* $\mu_A(x) = 1$.

Tradotto in un modello matematico diventa un problema di ottimizzazione, 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 di scarto legata al posizionamento dei punti all'interno dell'ipersfera mentre τ è la variabile di scarto 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 di x_i da a ($\|\phi(x_i) - a\|^2$) è minore o uguale al raggio dell'ipersfera ($R^2 + \xi_i$) e quindi è posizionato interamente ad 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).

Più conveniente è risolvere il problema sopra citato tramite il suo duale. Formalmente otteniamo:

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

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

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

Risolvendo il duale, si può dimostrare che, preso qualunque punto $x \in X$ ed una funzione kernel k, vale:

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

il che ci permette di calcolare la distanza tra il centro a dell'ipersfera nell'iperspazio generato da k nella soluzione ottimale e $\phi(x)$.

Analogamente al Support Vector Clustering in *possibilearn* si individuano i *support vector*, ossia i punti che delimitano la zona del cluster 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, definendo $R^{*^2}(x)$ come la distanza di x dal centro dell'ipersfera nella soluzione ottimale, possiamo riscrivere quanto appena detto come:

$$R^{*^2}(x) \leq R^{*^2}(s) \quad \forall x \mid \mu(x) = 1 \wedge \forall s \in S,$$

dove S definisce l'insieme di tutti i support vector dell'ipersfera.

Tutto quello che succede per i punti esterni all'ipersfera dipende da funzioni chiamate *fuzzifier*. Le funzioni *fuzzifier* possono assumere qualunque forma e descrivono come si comporta $\mu(x)$ al crescere della distanza di x dall'ipersfera nell'iperspazio generato dal kernel k. Queste funzioni, come la costante C ed il kernel k definiti precedentemente, vengono definiti iperparametri.

2.3.2 Configurazione degli iperparametri

Iperparametro è un termine 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 semplici 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*:

- Linear fuzzifier ($\hat{\mu}_{\text{lin}}$),
- Crisp fuzzifier ($\hat{\mu}_{\text{crisp}}$),
- Quantile constant piecewise fuzzifier ($\hat{\mu}_{\text{qconst}}$),
- Quantile linear piecewise fuzzifier ($\hat{\mu}_{\text{qlin}}$).

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

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

Questa costante rappresenta un iperparametro che definisce il costo dell'errore nella classificazione degli elementi durante la costruzione dell'ipersfera. Quesa C funziona in maniera analoga alla C nelle SVM (Support Vector Machine). Tanto più è grande C, tanto più sarà elevato il costo dell'errore nella classificazione della

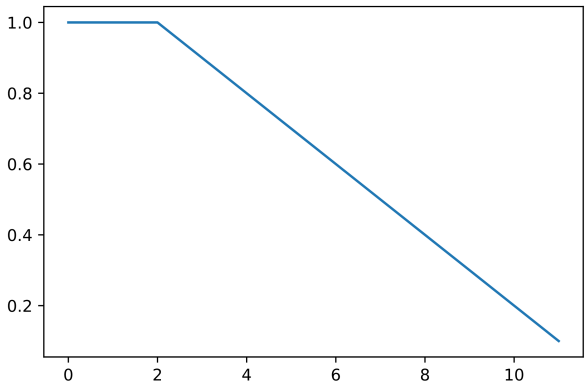
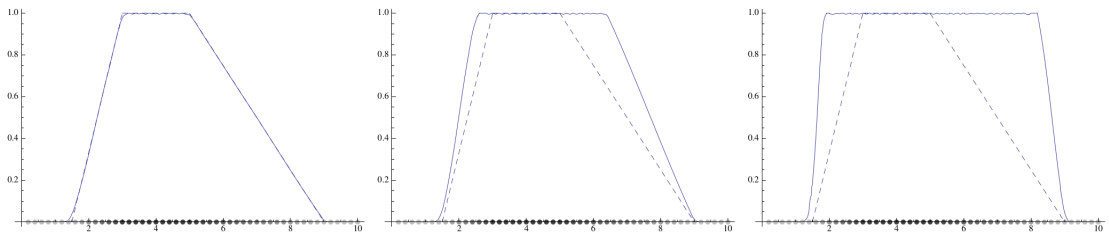


Figura 13: Esempio di Linear fuzzifier ($\hat{\mu}_{\text{lin}}$) per un insieme I. L'asse delle ordinate rappresenta il grado di appartenenza e l'asse delle ascisse rappresenta il valore della *feature*. Al crescere della *feature* diminuisce il grado di appartenenza all'insieme I in modo lineare.

variabile i-esima. Al contrario, tanto più è piccola C, tanto più piccolo sarà il costo dell'errore.

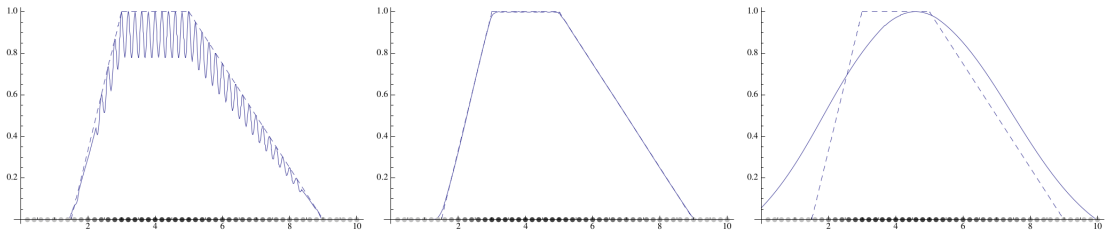
In Figura 14, possiamo osservare il ruolo di questo iperparametro al variare del suo valore. In ciascun grafico della Figura al variare di C osserviamo come si modifica una stessa funzione di appartenenza $\hat{\mu}$. Sapendo che la funzione obiettivo che vogliamo risolvere è una funzione di minimo, bisogna mantenere il costo più basso possibile. Questo si traduce visivamente in una funzione "rigida" avente un cambiamento molto brusco. È facile notare che maggiore è C e maggiore sarà la somiglianza di $\hat{\mu}$ ad una "funzione di appartenenza" in un insieme standard.



(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

Esiste un ulteriore parametro che modifica la forma del *fuzzifier*, questo è il kernel. Il kernel, come abbiamo visto nelle SVM è una funzione che viene utilizzata per mappare uno spazio N-dimensionale, in uno spazio M-dimensionale (dove M è spesso molto più grande di N). Esso compare nella ridefinizione della distanza di $R^2(x)$. Questo iperparametro permette di modificare la forma di $\hat{\mu}$. Prendendo l'esempio in Figura 15 e una funzione kernel gaussiana, vediamo come modificando σ (parametro della funzione kernel gaussiana) viene modificata la forma della funzione di appartenenza.



(d) $\sigma = 0.075$ (e) $\sigma = 0.12$ (f) $\sigma = 2$

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

Ricapitolando in *possibilearn* osserviamo tre diversi iperparametri:

- 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.

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

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

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

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