

Sistemi Autonomi

Implementazione e comparazione tra algoritmi di estrazione di conoscenza da reti neurali

Federico Sabbatini

836641

Black Boxes (BBs).

Sempre più spesso i sistemi di supporto alle decisioni vengono costruiti come *Black Boxes* (BBs), ovvero sistemi che nascondono all'utente utilizzatore la loro logica interna. In genere una BB sfrutta sofisticati modelli di *Machine Learning* per prevedere dati e risultati, a volte sensibili. In questi casi la mancanza di una spiegazione del risultato e del processo che ha portato alla sua produzione rende inutilizzabile la BB stessa. Si pensi, ad esempio, ad una applicazione critica in cui l'utente utilizzatore deve prendere una decisione cruciale in termini di sicurezza o la cui fallacia si esprima come un enorme svantaggio economico. È ovvio che tale utente non può fidarsi del responso della BB senza capire il ragionamento che ha portato ad esso: non è ammissibile effettuare una scelta "perché lo ha detto la BB".

Un'altra questione derivante dall'impossibilità di leggere i processi interni delle BBs è legata alla discriminazione automatica. Si è visto in passato, ad esempio, che è possibile utilizzare un programma per discriminare e penalizzare una certa classe di persone sulla base dell'appartenenza etnica o del sesso, pur senza chiedere informazioni al riguardo. Questo è possibile in quanto l'algoritmo è in grado di dedurre tali dati sfruttando nome e luogo di nascita delle persone. Se il modello non è ispezionabile, risulta quasi impossibile rilevare anomalie di questo genere.

Da questi problemi nasce quindi l'esigenza di estrarre la conoscenza dalle BBs, di interpretarle, ispezionarle, comprenderle, rappresentarle e riprodurle in maniera più *trasparente* rispetto al modello originale. Il concetto di *trasparenza* va appunto contrapposto a quello di *opacità*, caratteristica principale dei modelli BB.

Un vantaggio collaterale del processo di estrazione della conoscenza è la possibilità di avere a disposizione uno strumento che consente agli esseri umani di capire in corrispondenza di quali istanze di *input* la predizione della BB risulta essere errata e, di conseguenza, quando la generalizzazione del modello fallisce.

Interpretabilità dei modelli.

Tre delle caratteristiche più importanti che deve possedere un modello decisionale sono l'interpretabilità, la comprensibilità e la capacità di poter essere spiegato. Più

precisamente, l'interpretabilità di un modello consiste nell'avere l'abilità di spiegare o fornire un significato del risultato in termini comprensibili ad un umano.

L'interpretabilità ha tre dimensioni, ovvero: globalità/località a seconda che si possa interpretare, rispettivamente, l'intero modello oppure solo le singole decisioni; la limitazione temporale, cioè il tempo impiegato da un utente per capire la spiegazione; la natura della competenza umana, cioè tenere conto del *background* e dell'esperienza degli utenti utilizzatori.

Caratteristiche ideali e tipologie di modelli interpretabili.

Un modello interpretabile deve tenere conto della seguente lista di caratteristiche:

- Interpretabilità: quanto il modello o le predizioni sono comprensibili per gli umani;
- Accuratezza: in che misura il modello predice in maniera precisa istanze del problema mai viste prima;
- Fedeltà: quanto il modello riesce a imitare un predittore di tipo BB. Si calcola come l'accuratezza, ma utilizzando come confronto non l'*output* atteso, bensì quello della BB.

Esistono diversi tipi di modelli interpretabili, dotati di differente complessità sia in termini di costruzione che di comprensibilità dal punto di vista umano. Un sistema decisionale basato su un albero decisionale sfrutta un grafo strutturato come un albero composto da nodi interni che rappresentano condizioni sulle *features* degli oggetti del *data set* e da foglie che indicano le etichette delle classi. Ogni ramo rappresenta un possibile percorso che porti da *input* ad *output*. I percorsi dalla radice alle foglie rappresentano le regole decisionali, pertanto l'albero può essere generalizzato in una serie di regole di tipo *if-then*.

Una regola decisionale è una funzione che mappa un'osservazione a un'azione appropriata. Tali regole possono essere estratte generando le cosiddette regole di classificazione. Le più comuni sono le *if-then* (formate da congiunzioni, disgiunzioni o negazioni applicate alle variabili di *input*). Altre regole (più compatte o generali) sono le *M-of-N* (vere se sono verificate *M* delle *N* condizioni esplicitate) oppure le *list of rules* (si considera solo la prima che regola verificata), le *falling rule lists* oppure i *decision sets*.

Tipi di problemi relativi all'analisi delle BBs.

L'analisi di una BB può dividersi in due categorie principali: *reverse engineering* oppure *design* delle spiegazioni. Nel primo caso si cerca di ricostruire una spiegazione della BB partendo dall'insieme delle sue decisioni. Nel secondo caso, invece, si parte da un insieme di decisioni di addestramento per sviluppare un modello predittivo interpretabile insieme alle spiegazioni. In queste due categorie rientrano quattro tipi di problemi: *model explanation*, *outcome explanation* e *inspection problem* nella prima, *transparent design* nella seconda.

Il *black box model explanation problem* consiste nel fornire, dato un modello BB che risolve problemi di classificazione, un modello interpretabile e trasparente in grado di imitarlo. Il modello costruito risulta essere globalmente interpretabile.

Il *black box outcome explanation problem* fornisce un *output* interpretabile, cioè associa alla predizione una spiegazione. Il modello costruito risulta essere localmente interpretabile.

Il *black box inspection problem* consiste nel fornire una rappresentazione che aiuti a capire come funziona la BB o perché restituisce certi dati anziché altri.

Il *transparent box design problem* consiste nel fornire un modello globalmente o localmente interpretabile dati una BB e un problema di classificazione.

Artificial Neural Network (ANN).

Quando si parla di rete neurale artificiale (ANN) si intende un modello computazionale ispirato alla rete neurale biologica. In tale modello si hanno degli strati di neuroni (di *input*, di *output* e nascosti o *hidden*) in cui ogni neurone è connesso a quelli dello strato immediatamente precedente e immediatamente successivo. Ogni connessione ha un parametro chiamato peso e ogni neurone ha un ulteriore parametro chiamato *bias*, indipendente dalle connessioni e dai pesi. Nel modello *feed forward* la propagazione dei dati avviene dallo strato di *input* verso quello di *output*, passando per gli eventuali strati nascosti. Ogni neurone propaga un valore di uscita calcolato applicando una funzione di attivazione al valore totale che legge in ingresso. Il valore di ingresso $net(i)$ del neurone i appartenente allo strato k si ottiene moltiplicando l'*output* dei neuroni dello strato

precedente $k-1$ connessi a i per il peso di ciascuna connessione e sommando tutti questi prodotti al *bias* di i .

In un problema di classificazione è frequente utilizzare reti neurali con tanti neuroni nello strato di *input* quante sono le *features* a disposizione del classificatore, un neurone nello strato di *output* per ogni classe del problema e un certo numero di strati nascosti che dipende dalla natura del problema. L'assenza di uno strato nascosto permette alla rete di rappresentare unicamente decisioni e funzioni linearmente separabili. Un solo strato consente di approssimare qualsiasi funzione che contenga un *mapping* continuo da uno spazio finito a un altro. Aumentando il numero degli strati si possono apprendere funzioni sempre più complesse con gradi di accuratezza sempre maggiori. Il numero di neuroni negli strati nascosti rappresenta un altro problema critico nella scelta dell'architettura di una rete. Impiegare pochi neuroni implica incorrere in problemi di *underfitting*, cioè nell'impossibilità di rilevare adeguatamente i segnali in un *data set* complesso. Viceversa, usare troppi neuroni può causare *overfitting*. In questo caso la rete ha così tanta capacità di *processing* che il *training set* utilizzato per addestrarla è insufficiente per addestrare tutti i neuroni, senza contare che un maggior numero di neuroni implica tempi maggiori di addestramento. Esistono diverse regole per determinare un numero adeguato di neuroni, non tutte applicabili assieme in quanto alcune di esse sono in contraddizione. Alcuni esempi sono i seguenti: il numero di neuroni dello strato nascosto deve essere compreso tra quello dello strato di *input* e quello dello strato di *output*; oppure deve essere due terzi la dimensione dello strato di *input* più la dimensione di quello di *output*; etc... Queste ed altre regole possono essere utilizzate come punto di partenza da raffinare mediante tecniche *trial and error*.

Le reti neurali sono delle BBs a tutti gli effetti: immagazzinano la conoscenza sotto forma di topologia e parametri (ovvero pesi e *biases*) ed è impossibile osservarle e trarre conclusioni senza adottare una qualche tecnica di estrazione della conoscenza.

Estrazione della conoscenza da reti neurali.

Nell'ambito delle ANN il processo di estrazione della conoscenza avviene mediante due diverse metodologie, chiamate *decomposizionale* e *pedagogica*. A queste due si aggiunge una terza modalità, detta *eclettica*, che combina aspetti di entrambe.

L'approccio decomposizionale estrae regole a livello delle singole unità della rete (neuroni degli strati nascosti e di *output*). Effettua delle considerazioni sui valori ricevuti in ingresso dalle unità, cercando di stabilire in quali casi un neurone risulta essere attivo o meno,

considerandoli come oggetti la cui uscita è un valore binario (sì oppure no). Di conseguenza, queste metodologie vedono la rete neurale come un modello trasparente.

Viceversa, l'approccio pedagogico vede la ANN come un modello opaco, ovvero una BB vera e propria. L'obiettivo è quello di trovare una funzione che mappi gli *input* della rete e i relativi *output* prodotti dalla rete, ovvero una funzione che approssimi nella maniera più accurata possibile la ANN. I parametri della rete non hanno nessun ruolo in questo procedimento.

Entrambi gli approcci utilizzano la rete già addestrata. A seconda dell'algoritmo che si applica, potrebbe essere necessario addestrare la rete in maniera particolare.

Rule extraction as learning.

Rule extraction as learning è un metodo eclettico che produce una lista di regole per ogni classe riconosciuta dalla rete neurale. Le regole possono essere sia di tipo *if-then* che *M-of-N*. L'algoritmo generale viene schematizzato dagli autori Craven e Shavlik nel loro articolo "*Using sampling and queries to extract rules from trained neural networks*" secondo lo schema seguente:

```
/* initialize rules for each class */
for each class  $c$ 
     $R_c := \emptyset$ 
repeat
     $e := \text{EXAMPLES}()$ 
     $c := \text{classify}(e)$ 
    if  $e$  not covered by  $R_c$  then
        /* learn a new rule */
         $r := \text{conjunctive rule formed from } e$ 
        for each antecedent  $r_i$  of  $r$ 
             $r' := r$  but with  $r_i$  dropped
            if  $\text{SUBSET}(c, r') = \text{true}$  then  $r := r'$ 
         $R_c := R_c \vee r$ 
until stopping criterion met
```

Ad ogni classe c del classificatore è associato un insieme di regole R_c . Finché non viene soddisfatto un determinato criterio di arresto si provvede ad eseguire le seguenti operazioni: selezionare un esempio e (estraendolo dal *training set* oppure producendolo in

maniera coerente rispetto alle caratteristiche del *data set*) e calcolare la classe c che viene restituita dalla rete neurale per l'esempio e . Se l'esempio e non è coperto dalle regole di R_c allora si procede con l'acquisizione di una nuova regola. Per prima cosa si forma la regola r come congiunzione delle *features* di e e successivamente si procede a generalizzare r . Infine la regola r viene aggiunta all'insieme R_c mettendola in disgiunzione rispetto alle regole già presenti. Il processo di generalizzazione della regola consiste nel determinare quali atomi della formula sono irrilevanti ai fini della classificazione. Per fare ciò si esegue un ciclo su ogni atomo ri della regola e per ciascuno si crea una nuova regola più generale r' data da r senza l'atomo ri . Se r' appartiene ancora alla classe c (e di conseguenza la funzione *subset* restituisce *true*), allora r viene generalizzata e assume il valore di r' , poi si continua con gli altri atomi, se ci sono.

L'algoritmo appena descritto produce una lista di *if-then rules*, ma gli autori propongono anche un algoritmo alternativo che restituisce invece una lista di regole di tipo *M-of-N*. Il vantaggio di queste ultime è che descrivono meglio il comportamento di una rete neurale, nel senso che ne servono meno per esprimere il funzionamento del modello e sono più compatte (e quindi di più facile lettura per gli umani).

Una limitazione dell'algoritmo riportato nell'articolo è che considera solo *features* categoriche; tuttavia la maggior parte dei *data set* contengono anche campi continui.

TREPAN.

Gli stessi autori propongono, nell'articolo "*Extracting tree-structured representations of trained networks*", un algoritmo alternativo per estrarre alberi decisionali dalle reti neurali. Tale algoritmo, *TREPAN*, ha diversi vantaggi: dal punto di vista della complessità computazionale è molto più performante, a parità di *BB* e *data set* presenta una fedeltà maggiore e le regole che fornisce in *output* sono molto più leggibili (anche in virtù del fatto che sono un numero minore). L'algoritmo schematizzato nell'articolo degli autori è riportato nella pagina seguente.

L'algoritmo costruisce un albero in cui ad ogni nodo corrisponde un'etichetta e un insieme di esempi. L'etichetta è data dalla classe dominante tra gli esempi (quella con maggior numero di occorrenze). Inoltre ogni nodo ha una priorità data dalla combinazione della sua raggiungibilità (il numero di esempi che contiene rispetto al numero totale degli esempi) e

della sua fedeltà (il numero di esempi appartenenti alla classe dominante rispetto a tutti gli esempi del nodo).

```

TREPAN(training_examples, features)
  Queue :=  $\emptyset$                                 /* sorted queue of nodes to expand */
  for each example  $E \in \text{training\_examples}$       /* use net to label examples */
    class label for  $E := \text{ORACLE}(E)$ 
  initialize the root of the tree,  $T$ , as a leaf node
  put  $\langle T, \text{training\_examples}, \{\} \rangle$  into Queue
  while Queue is not empty and  $\text{size}(T) < \text{tree\_size\_limit}$  /* expand a node */
    remove node  $N$  from head of Queue
     $\text{examples}_N :=$  example set stored with  $N$ 
     $\text{constraints}_N :=$  constraint set stored with  $N$ 
    use features to build set of candidate splits
    use  $\text{examples}_N$  and calls to  $\text{ORACLE}(\text{constraints}_N)$  to evaluate splits
     $S :=$  best binary split
    search for best  $m$ -of- $n$  split,  $S'$ , using  $S$  as a seed
    make  $N$  an internal node with split  $S'$ 
    for each outcome,  $s$ , of  $S'$                       /* make children nodes */
      make  $C$ , a new child node of  $N$ 
       $\text{constraints}_C := \text{constraints}_N \cup \{S' = s\}$ 
      use calls to  $\text{ORACLE}(\text{constraints}_C)$  to determine if  $C$  should remain a leaf
      otherwise
         $\text{examples}_C :=$  members of  $\text{examples}_N$  with outcome  $s$  on split  $S'$ 
        put  $\langle C, \text{examples}_C, \text{constraints}_C \rangle$  into Queue
  return  $T$ 

```

Si può notare che l'algoritmo utilizza una coda con priorità per memorizzare i nodi ancora da espandere. I nodi vengono scelti in ordine di priorità proprio per espandere prima quelli più rilevanti. Questo è particolarmente utile nel caso in cui l'algoritmo dovesse terminare perché viene raggiunto il numero massimo di nodi consentito: in questo modo si evita di sprecare nodi espandendo quelli poco rilevanti.

L'espansione di un nodo avviene selezionando il migliore *M-of-N split* binario tra quelli consentiti: al nodo vengono aggiunti due nodi figli e gli esempi del nodo genitore vengono spartiti tra di essi. I nodi figli ereditano anche i vincoli del nodo genitore (cioè le scelte che consentono di raggiungere il nodo partendo dalla radice dell'albero).

L'algoritmo ha diversi parametri: oltre al numero massimo di nodi esiste anche un numero minimo di esempi che deve possedere un nodo prima di poter valutare il miglior *split*. Se un nodo non ha sufficienti esempi, è possibile sfruttare un oracolo che ne produca a piacere (rispettando i vincoli del nodo e con coerenza rispetto al *data set*). L'oracolo viene utilizzato anche per verificare se la generalizzazione di una regola continua ad appartenere alla stessa classe della regola non generalizzata.

Implementazione degli algoritmi.

Di seguito vengono presentate le implementazioni dell'algoritmo *rule extraction as learning* e di una versione semplificata di *TREPAN*. Il linguaggio di programmazione utilizzato per svilupparli è Python versione 3. Entrambi gli algoritmi sono stati estesi anche per l'elaborazione di *features* continue. Il *data set* utilizzato viene pre-processato in modo che ogni *feature* sia un valore di verità (0 se la caratteristica è assente o 1 se è presente). Per fare ciò le *features* categoriche vengono esplose in tante *features* quanti sono i possibili valori che possono assumere e ogni nuova *feature* avrà il valore 1 in corrispondenza di una sola colonna e 0 in tutte le altre, coerentemente con l'informazione codificata inizialmente nel *data set*. Analogamente, le *features* continue o numeriche discrete vengono trasformate in *features* booleane in cui ogni *feature* rappresenta un intervallo di valori. Un solo intervallo per ogni *feature* sarà contrassegnato dal valore 1, tutti gli altri saranno settati a 0, secondo l'informazione originale contenuta nel *data set*. Il numero di intervalli viene scelto automaticamente in base al numero di classi (un intervallo per ogni classe). Gli estremi di ogni intervallo vengono iterativamente raffinati finché l'unione di tutti gli intervalli copre l'intero spazio della *feature* dal valore minimo al valore massimo contenuto nel *data set*, mentre l'intersezione di ogni coppia di intervalli risulta essere vuota (intervalli disgiunti). L'obiettivo è quello di provare ad accoppiare intervalli e classi di *output*. Per fare ciò ogni intervallo viene inizializzato con estremi inferiore e superiore uguali al valore medio della distribuzione della *feature* relativamente ad una differente classe. Iterativamente si provvede ad ampliare gli intervalli a coppie, finché non si crea una sovrapposizione. Quando una sovrapposizione viene rilevata, l'espansione termina e ai due estremi viene assegnato il loro valore medio. L'espansione avviene iterativamente considerando la deviazione standard della distribuzione della *feature* di ogni classe, in modo da espandere maggiormente, a parità di iterazioni, intervalli che devono coprire uno spazio maggiore.

Al termine della fase di *pre-processing* si avrà, pertanto, un'esplosione del numero di *features*, in quanto ogni *feature* categorica che può assumere n valori produrrà n *features*, mentre ogni *feature* numerica produrrà m *features* se il classificatore è addestrato per riconosce m classi. Non è più necessario scalare il *data set* in quanto tutti i valori contenuti in esso sono 0 oppure 1.

Le immagini seguenti mostrano la conversione di *features* categoriche e continue in *features* booleane, secondo il procedimento descritto sopra. Inoltre viene mostrato un esempio di creazione automatica e affinamento iterativo di *ranges* per *features* continue.

Passaggio da *feature* categorica a *feature* booleana:

	...	Colore	...
A	...	Verde	...
B	...	Rosso	...
C	...	Blu	...



	...	Colore Verde	Colore Rosso	Colore Blu	...
A	...	1	0	0	...
B	...	0	1	0	...
C	...	0	0	1	...

Passaggio da *feature* continua a *feature* booleana:

	...	Lunghezza in cm	...
A	...	2,5	...
B	...	4,2	...
C	...	3,8	...



	...	Lunghezza in cm 2,5 - 3,35	Lunghezza in cm 3,35 - 4,2	...
A	...	1	0	...
B	...	0	1	...
C	...	0	1	...

Esempio di creazione automatica e raffinamento iterativo di intervalli. Si considerano il *data set "Iris"* e la *feature* continua "*petal length in cm*".

Inizialmente viene creato un *range* per ogni classe con estremo inferiore uguale all'estremo superiore, entrambi pari alla media dei valori della *feature* in corrispondenza di ciascuna classe. Si ricorda che il *data set* preso come esempio ha tre diverse classi: "*Iris Setosa*", "*Iris Versicolor*" e "*Iris Virginica*".

petal length in cm

```
Iris-setosa      (1.464, 1.464, 0.174, '-', '+')
Iris-versicolor (4.26, 4.26, 0.470, '-', '+')
Iris-virginica  (5.552, 5.552, 0.552, '-', '+')
```

I cinque parametri di ogni intervallo rappresentano, rispettivamente, estremo inferiore, estremo superiore, deviazione standard della distribuzione, direzione in cui si deve espandere l'estremo inferiore (inizialmente verso sinistra) e direzione in cui si deve espandere l'estremo superiore (inizialmente verso destra).

I primi due intervalli da far convergere sono quelli relativi a *Iris Setosa* e *Iris Versicolor*. In particolare si dovranno far convergere gli estremi evidenziati:

```
(1.464, 1.464, 0.174, '-', '+')
(4.26, 4.26, 0.470, '-', '+')
```

A tal fine, l'estremo superiore relativo alla classe *Iris Setosa* aumenterà ad ogni iterazione di 0.174, mentre l'estremo inferiore relativo all'*Iris Versicolor* ad ogni iterazione verrà ridotto di 0.470. Il ciclo si interrompe quando i due intervalli si sovrappongono. Nel caso specifico, servono nove iterazioni per verificare tale condizione:

(1.464, 1.638, 0.174, '-', '+')	(1.464, 2.158, 0.174, '-', '+')	(1.464, 2.679, 0.174, '-', '+')
(4.086, 4.26, 0.470, '-', '+')	(3.566, 4.26, 0.470, '-', '+')	(3.045, 4.26, 0.470, '-', '+')
(1.464, 1.811, 0.174, '-', '+')	(1.464, 2.332, 0.174, '-', '+')	(1.464, 2.852, 0.174, '-', '+')
(3.913, 4.26, 0.470, '-', '+')	(3.392, 4.26, 0.470, '-', '+')	(2.872, 4.26, 0.470, '-', '+')
(1.464, 1.985, 0.174, '-', '+')	(1.464, 2.505, 0.174, '-', '+')	(1.464, 3.026, 0.174, '-', '+')
(3.739, 4.26, 0.470, '-', '+')	(3.219, 4.26, 0.470, '-', '+')	(2.698, 4.26, 0.470, '-', '+')

A questo punto viene annullata l'ultima espansione e vengono aggiornate le direzioni di espansione degli estremi. Infine ad ogni estremo viene assegnato il valore medio (si veda l'immagine seguente) e si passa al raffinamento della successiva coppia di estremi (*Iris Versicolor* e *Iris Virginica*).

$(1.464, 2.852, 0.174, '-', '-') \rightarrow (1.464, 2.862, 0.174, '-', '+')$
 $(2.872, 4.26, 0.470, '+', '+') \rightarrow (2.862, 4.26, 0.470, '-', '+')$

Analogamente per l'altra coppia di classi:

$(2.862, 4.26, 0.470, '-', '+') \rightarrow (2.862, 5.200, 0.470, '-', '+')$
 $(5.552, 5.552, 0.552, '-', '+') \rightarrow (4.612, 5.552, 0.552, '-', '+')$
 $(2.862, 4.730, 0.470, '-', '+') \rightarrow (2.862, 4.730, 0.470, '-', '-')$
 $(5.082, 5.552, 0.552, '-', '+') \rightarrow (5.082, 5.552, 0.552, '+', '+')$

Come risultato finale si avranno i seguenti intervalli:

$(2.862, 4.906, 0.470, '-', '+')$
 $(4.906, 5.552, 0.552, '-', '+')$

A questo punto i tre intervalli ottenuti sono disgiunti e adiacenti, tuttavia non coprono tutto lo spazio della *feature*. Per risolvere questo inconveniente vengono sostituiti l'estremo inferiore del primo intervallo e l'estremo superiore dell'ultimo intervallo rispettivamente con i valori minimo e massimo della *feature*. Al termine della correzione i tre intervalli saranno i seguenti:

$(1.000, 2.862)$
 $(2.862, 4.906)$
 $(4.906, 6.900)$

Implementazione delle classi ausiliarie.

Sia l'algoritmo *rule extraction as learning* che la versione semplificata di *TREPAN* utilizzano delle classi ausiliarie comuni per pre-processare il *data set* e per creare e addestrare una rete neurale tramite gli strumenti della libreria Python *Scikit-learn*. Tali classi sono *Dataset* e *Classifier*.

La classe *Dataset* espone dei metodi per leggere un *data set* da un *file* in formato CSV e per effettuare diverse operazioni sui dati (rimuovere le *features* categoriche, rimuovere le

features reali, scalare i dati, dividere il *data set* in *train set* e *test set*, visualizzare le statistiche del *data set*).

La classe *Classifier*, invece, espone dei metodi per creare e addestrare una rete neurale dotata di un numero di strati nascosti scelto dall'utente (ogni strato ha a sua volta un numero di unità scelto dall'utente). Inoltre consente la visualizzazione delle statistiche riguardanti l'accuratezza della rete.

Ad esempio, se si volesse leggere il *data set* contenuto nel file "iris.data" è sufficiente utilizzare il seguente codice:

```
# caricamento del dataset
dataset = Dataset("iris")
# conversione delle feature categoriche in feature booleane
dataset.removeCatCols()
# conversione delle feature continue in feature booleane
dataset.removeContCols()
# suddivisione del dataset in insieme di training e testing
Xtrain, Xtest, ytrain, ytest = dataset.splitData()
```

Come risultato dell'esecuzione si ottengono quattro vettori, rispettivamente un *data set* con cui effettuare l'addestramento della rete, uno con cui effettuare il *testing* e i due vettori contenenti le classi corrette dei due *data sets*.

I vettori ottenuti possono essere utilizzati per addestrare una rete con il seguente codice:

```
# scelta automatica del numero di neuroni dell'hidden layer
# tenendo conto del numero di feature e del numero di classi
hiddenSize = int(len(dataset.features()) / 3 + len(dataset.classes()))
# creazione di una rete neurale per effettuare la classificazione
mlp = Classifier(hiddenSize, 5000)
# addestramento della rete
mlp.train(Xtrain, ytrain)
# stampa dei dati relativi all'accuratezza della rete
mlp.accuracy(Xtest, ytest)
```

Di *default* viene creata una rete neurale con un solo *hidden layer* dotato di un numero di neuroni pari a un terzo del numero delle *features* del *data set* più il numero delle differenti classi. Utilizzando questa formula si ottiene un numero di neuroni sufficiente per ottenere una buona accuratezza e che al contempo consenta di addestrare la rete in tempi ragionevoli. Aumentare tale numero non incide in maniera evidente sull'accuratezza della rete, ma peggiora drammaticamente i tempi di elaborazione. Diminuire troppo tale numero, invece, impatta negativamente sull'accuratezza.

L'*output* del metodo *accuracy* è il seguente:

Confusion matrix:

```
[[ 8  0  0]
 [ 0  9  1]
 [ 0  1 11]]
```

Classification report :

	precision	recall	f1-score	support
Iris-setosa	1.00	1.00	1.00	8
Iris-versicolor	0.90	0.90	0.90	10
Iris-virginica	0.92	0.92	0.92	12
accuracy			0.93	30
macro avg	0.94	0.94	0.94	30
weighted avg	0.93	0.93	0.93	30

Il *data set* estratto e la rete neurale addestrata sono il punto di partenza per i due algoritmi descritti in seguito.

Implementazione di *rule extraction as learning*.

Rule extraction as learning è stato implementato esattamente come descritto in precedenza. Tutto ciò che riguarda l'algoritmo è stato incapsulato nella classe *RuleSet*, che espone metodi per estrarre le regole, calcolare la fedeltà del modello estratto rispetto alla rete originale e stampare la lista di regole estratte.

Le tre operazioni possono essere eseguite in sequenza utilizzando il seguente codice:

```
# creazione di un oggetto di tipo RuleSet
RS = RuleSet(dataset, mlp)
# estrazione delle regole e creazione di una lista di regole
RS.extract(Xtrain)
# calcolo della fedeltà della lista di regole
RS.check(Xtest)
# stampa a video della lista
RS.printRules()
```

La fase di estrazione segue fedelmente l'algoritmo descritto dagli autori, con l'unica differenza che l'oracolo si limita a restituire gli esempi del *train set*, per evitare di usare esempi del *test set*. La fase di calcolo della fedeltà (metodo *check*) consiste nell'utilizzare ciascun esempio del *test set* per confrontare la classe restituita dalla rete neurale con quella restituita dalla lista di regole. Le regole sono formate in modo tale che una sola possa soddisfare un esempio. In tal caso l'esempio viene classificato con la classe relativa alla regola. Se però l'algoritmo non è riuscito a generalizzare sufficientemente le regole, allora nessuna regola soddisferà l'esempio.

Il metodo *printRules* stampa a video l'elenco delle regole. Per ogni classe viene mostrata una lista di regole da considerare in disgiunzione, mentre ogni regola è composta da due sequenze di atomi. La prima contiene gli atomi che devono essere veri, mentre la seconda quelli che devono essere falsi. La generalizzazione è data dall'assenza di alcuni atomi, ovvero da tutti quegli atomi il cui valore non incide sulla classe che viene attribuita alla regola, siano essi veri oppure falsi.

Per ridurre il numero di atomi (e come vantaggio collaterale, di conseguenza, ridurre il numero di regole e aumentare la generalizzazione) si tiene in considerazione il fatto che pre-processare il *data set* come descritto in precedenza comporta la creazione di n nuove *features* al posto di ogni vecchia *feature* categorica o numerica. Di queste n *features* solo una avrà il valore 1 (cioè sarà verificata), mentre le altre $n - 1$ saranno settate a 0 (cioè saranno false). Questa considerazione consente di eliminare da ogni regola tutti gli atomi nella lista di quelli falsi a patto che nella lista di quelli veri compaia il duale. Ad esempio, se tra gli atomi veri esiste una condizione del tipo "Colore = rosso", automaticamente si possono eliminare condizioni del tipo "Colore = verde" e "Colore = blu" dalla lista degli atomi falsi, perché è ovvio che la veridicità della condizione "Colore = rosso" implica la falsità delle altre due.

L'output dei metodi *check* e *printRules* viene mostrato di seguito:

```
Fidelity rate: 100.00%
List of rules:
Class is 'Iris-setosa' if:
|-- {petal width in cm 0.10-0.78} is TRUE and {petal length in cm 2.86-4.91, sepal length in cm 6.26-7.90, petal length in cm 4.91-6.90, sepal width in cm 2.00-2.87, sepal length in cm 5.47-6.26} is FALSE
|-- {petal length in cm 1.00-2.86} is TRUE and {petal width in cm 0.78-1.68, sepal length in cm 5.47-6.26, sepal length in cm 6.26-7.90, petal width in cm 1.68-2.50} is FALSE
|-- {petal length in cm 1.00-2.86} is TRUE and {sepal width in cm 2.00-2.87, petal width in cm 0.78-1.68, sepal length in cm 6.26-7.90, sepal width in cm 2.87-3.20} is FALSE
Class is 'Iris-versicolor' if:
|-- {sepal length in cm 6.26-7.90} is TRUE and {petal length in cm 1.00-2.86, petal length in cm 4.91-6.90, sepal width in cm 2.00-2.87, sepal width in cm 2.87-3.20, petal width in cm 1.68-2.50} is FALSE
|-- {petal length in cm 2.86-4.91} is TRUE and {petal width in cm 1.68-2.50, petal width in cm 0.10-0.78} is FALSE
|-- {sepal width in cm 3.20-4.40, petal length in cm 2.86-4.91} is TRUE and {sepal length in cm 6.26-7.90, petal width in cm 0.10-0.78} is FALSE
Class is 'Iris-virginica' if:
|-- {petal length in cm 4.91-6.90} is TRUE and {sepal width in cm 3.20-4.40} is FALSE
|-- {petal width in cm 1.68-2.50} is TRUE and {sepal width in cm 3.20-4.40, sepal length in cm 4.30-5.47, petal length in cm 1.00-2.86} is FALSE
|-- {petal width in cm 1.68-2.50} is TRUE and {sepal length in cm 4.30-5.47, petal length in cm 1.00-2.86, petal length in cm 2.86-4.91} is FALSE
|-- {sepal width in cm 2.00-2.87, petal width in cm 1.68-2.50} is TRUE and {petal length in cm 1.00-2.86} is FALSE
```

Implementazione di una versione semplificata di *TREPAN*.

Si è deciso di implementare una versione semplificata rispetto all'algoritmo *TREPAN* descritto dagli autori. Analogamente a *rule extraction as learning* tutte le funzionalità relative all'algoritmo (estrazione delle regole, calcolo della fedeltà e stampa dell'albero decisionale) sono state incapsulate nella classe *Trepan*.

Il seguente codice mostra in che modo è possibile eseguire le tre operazioni sequenzialmente:

```
# creazione di un oggetto di tipo Trepan
trepan = Trepan(Xtrain, dataset.same(), mlp)
# estrazione delle regole e creazione di un albero decisionale
trepan.compute()
# calcolo della fedeltà dell'albero
trepan.check(Xtest)
# stampa a video dell'albero estratto
trepan.printTree()
```

Il metodo *compute* è quello che si occupa di estrarre la conoscenza dalla rete, creando un albero decisionale. Per prima cosa si provvede a creare la radice dell'albero, a cui viene assegnato l'intero *training set* e nessun vincolo. Successivamente si provvede ad espandere questo nodo andando a selezionare su quale *feature* tra tutte le possibili conviene eseguire uno *split* binario. Durante l'analisi della radice le *features* possibili sono tutte; nell'analisi di nodi più profondi, invece, andranno rimosse tutte quelle già analizzate durante il percorso che porta dalla radice al nodo in esame. Inoltre, analogamente a *rule extraction as learning*, si provvede ad escludere l'analisi di tutte quelle *features* che sono banalmente false in quanto nel percorso dalla radice al nodo esiste un vincolo che imponga la veridicità di una *feature* duale (riprendendo l'esempio di prima, se nel nodo padre si è effettuato uno *split* sulla condizione "Colore = rosso", il ramo che identifica "Colore = rosso VERO" evita di analizzare le condizioni "Colore = verde" e "Colore = blu" per effettuare *splits* successivi, in quanto nessuno degli esempi che contiene rende vera una delle due condizioni).

La scelta del migliore *split* avviene tramite confronto delle fedeltà dei figli ottenuti applicando ogni possibile *split*. Viene selezionato lo *split* che massimizza la fedeltà media dei figli. Esistono delle considerazioni aggiuntive che possono essere fatte per migliorare l'algoritmo e pilotarlo verso *splits* più desiderabili di altri. Ad esempio, uno *split* che riesca a far sì che uno dei due figli abbia una classe in meno rispetto al nodo padre è una configurazione desiderabile, in quanto riesce ad isolare una classe in un solo ramo

dell'albero. Questo significa che tutte le considerazioni effettuate nell'altro ramo sono prive di interesse per la definizione di quali esempi appartengono a quella data classe.

Viceversa, uno *split* che generi due nodi figli in cui la classe prevalente è la medesima è poco desiderabile, in quanto appare come una decisione poco discriminante rispetto alle altre possibili.

Quando viene scelto uno *split* su una data condizione si provvede a dividere l'insieme di esempi del nodo esaminato in due sottoinsiemi (gli esempi con la condizione vera e quelli con la condizione falsa). Ogni sottoinsieme viene assegnato a un nodo figlio, in aggiunta ai vincoli del nodo padre e ad un nuovo vincolo (condizione vera oppure condizione falsa).

Ad ogni nodo, oltre a esempi e vincoli, vengono assegnate una classe (uguale alla classe più frequente all'interno dell'insieme degli esempi) e una priorità, come descritto in precedenza durante l'analisi dell'algoritmo *TREPAN* originale.

L'algoritmo sfrutta una coda prioritaria in cui vengono inseriti tutti i nodi che ancora devono essere espansi. Inizialmente contiene solo la radice, poi mano a mano che i nodi vengono prelevati ed espansi si provvede ad inserirvi i nodi figli ottenuti durante l'espansione.

L'algoritmo termina quando la coda dei nodi da espandere è vuota (ovvero tutti i nodi foglia contengono esempi appartenenti solo ad un'unica classe).

Dopo aver creato l'albero è possibile effettuare delle ottimizzazioni in modo da comprimere il numero di nodi (come vantaggio collaterale si ottiene una generalizzazione leggermente migliore). L'ottimizzazione avviene in due passi. Per prima cosa si provvede ad incorporare i nodi foglia all'interno dei relativi padri nel caso in cui abbiano la medesima classe dominante. Il secondo passo è dato dalla contrazione di diversi nodi interni consecutivi lungo un ramo e che hanno la stessa classe dominante in un unico nodo interno (con la stessa classe) che ha come vincoli la sequenza di tutti i vincoli dei nodi compressi. Questa operazione è fattibile solo se tutti i nodi della sequenza hanno un solo figlio (il secondo figlio è stato eliminato durante il primo passo dell'ottimizzazione).

La classificazione di un esempio consiste nel partire dalla radice e scegliere di volta in volta un ramo anziché un altro in base alla condizione discriminante del nodo e ai valori delle *features* dell'esempio. Com'è ovvio che sia, vengono percorsi i rami le cui condizioni soddisfano l'esempio. Nel caso in cui un nodo abbia un solo figlio la cui condizione non soddisfa l'esempio, il processo termina e la classe assegnata all'esempio è quella dell'ultimo nodo raggiunto. Se il ramo viene percorso fino ad una foglia, viene restituita la classe dominante della foglia. Se il ramo contiene un nodo con una sequenza di vincoli ottenuta dalla contrazione di più nodi, valgono le seguenti considerazioni: se il primo

vincolo della sequenza non è soddisfatto, la classe dell'esempio è quella del nodo padre; se sono soddisfatti tutti i vincoli della sequenza si continua la classificazione passando ai nodi figli; se è soddisfatto il primo vincolo ma non tutti gli altri il processo termina e viene restituita la classe del nodo.

L'*output* dei metodi *check* e *printTree* viene mostrato di seguito:

```
Fidelity: 100.00%
root > Iris-setosa
└─ !petal length in cm 1.00-2.86 > Iris-virginica
    └─ !petal length in cm 4.91-6.90 > Iris-versicolor
        └─ !petal width in cm 0.78-1.68 > Iris-virginica
            └─ sepal width in cm 3.20-4.40 > Iris-versicolor
```

Lettura dei risultati.

Mentre è incomprensibile (per un umano) interpretare le regole della rete neurale osservando i suoi parametri, è invece possibile confrontare la leggibilità della lista di regole e dell'albero decisionale.

Prendendo in esame la lista riportata in precedenza, si può ottenere la seguente lettura parziale:

- La classe è "iris setosa"
 - SE:
 - l'altezza dei petali è compresa tra 0.1 e 0.78 cm
 - MA:
 - la lunghezza dei petali NON è compresa tra 2.86 e 4.91 cm E
 - la lunghezza dei sepali NON è compresa tra 6.26 e 7.9 cm E
 - la lunghezza dei petali NON è compresa tra 4.91 e 6.9 cm E
 - l'altezza dei sepali NON è compresa tra 2 e 2.87 cm E
 - la lunghezza dei sepali NON è compresa tra 5.47 e 6.26 cm
 - OPPURE SE:
 - la lunghezza dei petali è compresa tra 1 e 2.86
 - MA:
 - ...
 - OPPURE SE ...

Come si può notare, anche partendo da un *data set* elementare si ottiene una lettura piuttosto confusa, in quanto per ogni classe ci sono molte regole e ogni regola ha molti atomi. L'albero offre una visione più chiara e lineare dello stesso problema di classificazione.

La lettura completa dell'albero estratto dalla medesima rete è la seguente:

- La maggior parte della popolazione appartiene alla classe "iris setosa";
- gli individui che NON hanno i petali di lunghezza compresa tra 1 e 2.86 cm, però, hanno molte probabilità di appartenere alla classe "Iris virginica";
- se però i petali NON sono neanche di lunghezza compresa tra 4.91 e 6.9 cm allora la classe più probabile è "iris versicolor";
- se in aggiunta l'altezza dei petali NON è compresa tra 0.78 e 4.68 cm allora la classe più probabile è "iris virginica";
- infine, se è vero anche che i sepali hanno un'altezza compresa tra 3.2 e 4.4 cm, la classe più probabile è "iris versicolor".

Confronto tra le implementazioni.

Di seguito vengono riportati i valori medi delle fedeltà ottenuti per entrambi gli algoritmi su diversi *data set*. La fedeltà rispetto alla rete neurale è criticamente correlata alla fedeltà della rete neurale rispetto alle classi reali del *data set*, anche se può sembrare strano in quanto ci si aspetta che un modello che imiti una rete neurale poco accurata sia molto fedele alla rete neurale ma non fedele alla realtà (o comunque fedele tanto quanto la rete). Invece se la rete è poco accurata, le regole estratte dalla rete sono ancora meno fedeli rispetto alla rete stessa. Probabilmente la scarsa accuratezza della rete dovuta a una generalizzazione poco efficiente induce delle situazioni in cui l'algoritmo di estrazione della conoscenza fa molta fatica a sua volta a generalizzare in maniera efficiente.

Nella tabella vengono anche riportati i seguenti valori: numero di esempi contenuti nel *data set*, numero di *features* prima e dopo il *pre-processing*, accuratezza della rete neurale, numero di nodi dell'albero decisionale prima e dopo l'ottimizzazione.

I campi contrassegnati con un asterisco indicano assenza di dati in quanto l'algoritmo è poco performante in termini di complessità computazionale e l'elaboratore non è riuscito a terminarlo con successo (terminazione per mancanza di memoria).

I valori medi sono stati calcolati su dieci esecuzioni per ciascun *data set*, tuttavia presentano una deviazione standard molto piccola.

<i>Data set</i>	<i>iris</i>	<i>car</i>	<i>wine</i>	<i>winequality-red</i>	<i>winequality-white</i>
N. esempi	150	1728	178	1599	4898
N. <i>features</i>	5	7	14	12	12
N. <i>features</i> dopo pre-processing	13	22	40	63	72
Accuratezza della rete neurale	95,0%	97,8%	95,8%	60,5%	53,5%
Fedeltà della lista di regole	95,1%	88,7%	*	*	*
Fedeltà dell'albero decisionale	99,2%	97,7%	94,6%	66,3%	68,2%
N. nodi dell'albero	9	205	57	1203	3411
N. nodi dell'albero dopo l'ottimizzazione	6	85	17	745	1989

Bibliografia.

- [1] R. Guidotti et al., “A Survey Of Methods For Explaining Black Box Models”, 2018.
- [2] R. Andrews, J. Diederich, A. B. Tickle, “Survey and critique of techniques for extracting rules from trained artificial neural networks”, *Knowledge-Based Systems*, vol. 8, pp. 373–389, 1995.
- [3] S. B. Thrun, “Extracting Provably Correct Rules from Artificial Neural Networks”, 1993.
- [4] M. W. Craven, J. W. Shavlik, “Using Sampling and Queries to Extract Rules from Trained Neural Networks”, *Proceedings of the Eleventh International Conference*, San Francisco, Stati Uniti, 10–13 luglio, 1994, pp. 37–45.
- [5] M. W. Craven, J. W. Shavlik, “Extracting Tree-Structured Representations of Trained Networks”, *Proceedings of the 8th International Conference on Neural Information Processing Systems*, Denver, Stati Uniti, 27 novembre – 2 dicembre, 1995, pp. 24–30.