Analisi del miglioramento delle performance ottenute tramite AdaBoost

Cristian Lepore
Corso di metodi statistici per l'apprendimento,
Laurea magistrale in Informatica
Dipartimento di Informatica
Università degli studi di Milano
Email: cristian.lepore@studenti.unimi.it
Mobile: +39 340 8071774

Abstract

In questo paper si mostrano i risultati su analisi del rischo ed overfitting, ottenuti grazie ad algoritmi di apprendimento parametrici e non. I classificatori proposti sono gli alberi di decisione, decision stumps e percettrone. Al fine di valutarne il miglioramento delle prestazioni tutti i predittori sono stati boostati con un algoritmo di ensemble learning. Come dataset per i test è stato scelto il Breast Cancer database, scaricabile liberamente dai repository UCI.

1 Introduzione

Immaginiamo di far partire una campagna pubblicitaria per un prodotto: per prima cosa scegliamo i dati sui clienti che hanno acquistato quel prodotto in passato. A questo punto costruiamo un profilo di persone potenzialmente interessate all'acquisto di quel prodotto ed una volta chiarito il target, ci sarà più semplice pianificare una campagna basandoci solo sui dati a disposizione. Volendo fare una teoria quindi, con il termine machine learning ci riferiamo all'individuazione automatica di modelli per l'estrazione di informazioni da una grossa quantità di dati.

Come noto, possiamo dividere le tecniche di apprendimento in supervisionate e non supervisionate. Negli esperimenti che seguono faremo riferimento sempre a tecniche supervisionate, vale a dire che in fase di addestramento confronteremo i modelli sulla base dell'etichetta reale dell'istanza. Al fine di ottenere una maggior accuratezza può essere una buona idea sfruttare metodi di ensemble learning, ad esempio il boosting. Questi meta-algoritmi costituiscono un buon metodo per ottenere un classificatore finale molto performante, estraendo interattivamente classificatori semplici (detti anche *weak learners*) ed associando loro il giusto peso nella votazione. Quello che ci aspettiamo di ottenere applicando questa tecnica con successo è una maggiore precisione nella classificazione delle istanze.

1.1 Scopo del lavoro

Lo scopo è valutare le performance di algoritmi di apprendimento diversi nel classificare istanze. Tutti i classificatori sono stati analizzati singolarmente e poi boostati con il meta-algoritmo Adaptive Boosting confrontando i risultati ottenuti.

Questo documento è organizzato come segue: nel *capitolo 2*, si descrivono i predittori utilizzati ed i principi di funzionamento dell'AdaBoost. Il *capitolo 3*, riporta le caratteristiche del dataset. Sperimentazione, metodo ed analisi correlate sono descritte nel *capitolo 4*. Infine l'*appendice* riporta la struttura dei files e cartelle all'interno del quale è

contenuto il codice che è stato sviluppato per i test.

2 Predittori utilizzati

Per l'analisi sono stati adoperati classificatori **parametrici** come il *percettrone*. Scelto perchè semplice ed economico, costruisce il classificatore in modo incrementale sfruttando tecniche greedy per ottenere il classificatore che minimizza il training error. Fornisce buone prestazioni soprattutto quando le istanze sono linearmente separabili. Come classificatore **non parametrico**, l'*albero di decisione* ci è sembrato una buona scelta. Esso costruisce incrementalmente un grafo aciclico, caratterizzato da una radice e da nodi foglia etichettati nel nostro caso con etichette binarie. Per mappare le istanze da un nodo al successivo si adopera una funzione test $t : x_i \rightarrow \{1,..,k\}$ che mappa il valore che assume l'istanza sull'attributo i-esimo in uno dei possibili nodi figli. Intuitivamente, l'albero di decisione dice quello che l'algoritmo ha appreso sul training set.
Entrambi, alberi di decisione e percettrone insieme con i *Decision Stumps*, possono essere utilizzati come weak learners all'interno di tecniche di ensemble learning come il Bagging o Boosting. Volendo possiamo vedere i decision stumps come alberi di decisione con un solo livello di profondità. Riassumendo, sono stati implementati i seguenti classificatori base:

1. Alberi di decisione
2. Decision Stumps
3. Percettrone

Al fine di creare un classificatore sofisticato questi classificatori base sono stati boostati con l'algoritmo di *Adaptive Boosting*.

2.1 Metodi di boosting

Il boosting più che un algoritmo è un modo di costruire algoritmi. Possiamo pensare al boosting come ad una tecnica per estrarre in modo incrementale membri da un comitato, farli votare ed assegnare loro il giusto peso. Il classificatore finale sarà dato dalla combinazione dei voti dei singoli membri. L'idea di base è quella di creare un modello che sbagli poco basandosi solamente su una sequenza di predittori semplici ed economici che overfittano poco.

AdaBoost

input:
training set $S = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$
weak learner WL
number of rounds T
initialize $\mathbf{D}^{(1)} = (\frac{1}{m}, \ldots, \frac{1}{m})$.
for $t = 1, \ldots, T$:
invoke weak learner $h_t = \mathrm{WL}(\mathbf{D}^{(t)}, S)$
compute $\epsilon_t = \sum_{i=1}^{m} D_i^{(t)} \mathbb{1}_{[y_i \neq h_t(\mathbf{x}_i)]}$
let $w_t = \frac{1}{2} \log\left(\frac{1}{\epsilon_t} - 1\right)$
update $D_i^{(t+1)} = \frac{D_i^{(t)} \exp(-w_t y_i h_t(\mathbf{x}_i))}{\sum_{j=1}^{m} D_j^{(t)} \exp(-w_t y_j h_t(\mathbf{x}_j))}$ for all $i = 1, \ldots, m$
output the hypothesis $h_s(\mathbf{x}) = \mathrm{sign}\left(\sum_{t=1}^{T} w_t h_t(\mathbf{x})\right)$.

Fig. 1. Pseudocodice per AdaBoost

2.1.1 AdaBoost

Come detto l'idea è quella di unire le votazioni di più weak learners al fine di creare un classificatore avente migliore accuratezza. L'output sarà dato dalla somma delle predizioni pesate dei singoli modelli, in accordo con la seguente formula:

$$f(x) = \sum_{i=1}^{T} w_i h_i(x)$$

Durante l'addestramento del modello, le istanze vengono ripesate e l'algoritmo darà maggior peso alle istanze misclassificate, nella speranza che il successivo modello sia più esperto su quest'ultime. In figura 1 riportiamo lo pseudocodice dell'algoritmo. Come si può notare ad ogni passo dell'algoritmo il classificatore base viene invocato, l'errore stimato ed un nuovo peso w_t calcolato. Ad ogni ciclo di iterazione il settaggio dei parametri ($\varepsilon_i \neq \frac{1}{2}$) e la nuova distribuzione di probabilità garantiscono un'opportuna scelta del prossimo classificatore base. Il ciclo di iterazione si arresta al raggiungimento del valore T, oppure se l'algoritmo non riesce a trovare un ε_i lontano da 0.5. Il rispetto di questa seconda condizione garantisce una rapida discesa del training error $\hat{er}(h)$.

2.2 Percettrone

Il percettrone è un classificatore lineare che costruisce modelli in modo incrementale che

possono essere visti come iperpiani del tipo: $w^T x - c = 0$. Fare training con il percettrone è semplice e l'aggiornamento del modello dipende dagli esempi misclassificati. Questi classificatori possono avere un errore di varianza basso, ma scommettono sul fatto che il bayesiano ottimo sia sufficientemente rigido altrimenti devono accontentarsi di commettere un errore di bias molto alto. In ogni caso la formula migliore che possiamo ottenere sulla convergenza del percettrone (valida per training set linearmente separabili e non) è questa:

$$M_T \leq \sum_{t=1}^{T} h_t(u) + (\|U\|X)^2 + \|U\|X\sqrt{\sum_{t=1}^{T} h_t(u)}$$

Significa che oltre a queste performance non possiamo andare. Qui M_T indica il numero di aggiornamenti che il percettrone deve fare prima di convergere, $h_t(u)$ è la funzione di perdita (Hinge Loss). U è il raggio della sfera che contiene tutti i nostri modelli ed X è il raggio della bolla che contiene le nostre istanze. Come detto la formula fornisce un maggiorante sul rischio che è pari al numero di aggiornamenti che dobbiamo compiere prima che il modello converga.

3 Motivazione

Vogliamo trovare una correlazione per le donne all'incidenza di tumori al seno benigni o maligni, sulla base dei risultati ad esami medici fatti in precedenza.

3.1 Dataset

Il dataset a disposizione è il *Wisconsin Breast Cancer*. Si tratta di dati forniti dall'Università del Wisconsin (US) e liberamente accessibili dal UCI machine learning repository. I dati si riferiscono a risultati acquisiti da ospedali su esami medici e vogliamo trovare una correlazione all'incidenza di tumori maligni sui soggetti presi in esame. Il dataset ad oggi conta 680 istanze contrapposte alle 367 che costituivano il dataset originario. Negli anni altri esempi hanno contribuito alla sua estensione fino al raggiungimento della dimensione odierna. Possiamo vedere ognuna di queste istanze come un vettore $V \in R^d$, cioè in uno spazio d-dimensionale con un numero di features $d = 9$. Ogni attributo varia in un range da 1-10 e deve essere immediatamente chiaro che rappresenta il risultato di un esame medico. L'ultima colonna contiene le etichette. Come detto nell'apprendimento supervisionato ogni esempio è dotato della sua etichetta che per comodità indichiamo con Y, contrapposta all'etichetta predetta dai nostri algoritmi $\hat{y}$. Ogni istanza appartiene a 2 possibili classi:

-1 per tumore benigno
+1 per tumore maligno

Riepilogando abbiamo 680 istanze con 9 attributi, ciascuno a rappresentare un esame clinico ed etichette binarie che assumono valore -1 (se tumore benigno) o +1 (se maligno).

3.1.1 Caratteristiche del dataset

Elenco delle features:

1. Spessore del coagulo
2. Dimensione delle celle
3. Forma delle cellula
4. Adesione marginale
5. Dimensione del Tessuto epiteliale
6. Nuclei
7. Cromatina
8. Nucleoli normali
9. Mitosi

Ciascuno di questi attributi assume valore numerico in un range [1-10].
Distribuzione delle classi:

Tumori benigni: 458 (65.5%)
Tumori maligni: 241 (34.5%)

Come detto i tumori benigni hanno etichetta -1; tumori maligni si trovano nel database con etichetta +1.
Il dataset varia in un range di valori interi con etichette binarie. Questo ci fa intuire di utilizzare gli algoritmi spiegati nella seconda sezione per poter risolvere problemi di classificazione binaria. La distribuzione delle classi sbilanciata (seppur di poco) a favore di una etichetta rispetto ad un'altra (65.5% vs 34.5%) suggerisce di implementare metriche di valutazione come *precision* e *recall*.

3.2 Software utilizzato

Tutto il codice è scritto in linguaggio R. R è un linguaggio di programmazione ed un ambiente di sviluppo specifico per l'analisi statistica dei dati. Distribuito nella sua versione standard con un'interfaccia a riga di comando, supporta diversi ambienti grafici (ad esempio Rstudio) che consentono di integrare R con nuovi pacchetti.
Scelto per la sua flessibilità e semplicità di utilizzo nel maneggiare dataset di grandi dimensioni, offre buone prestazioni anche quando si vuole lavorare con vettori e matrici.

4 Sperimentazione

Per i test il Breast Cancer dataset è stato suddiviso in due parti, training e test set. Come test si è scelta una porzione pari al 15% dell'intero dataset (102 esempi) e la rimanente parte è stata suddivisa tra training e validation set.
Per fare tuning dei parametri si è utilizzata la *Cross Validazione* che consiste nella suddivisione del training set in k parti di uguale numerosità da utilizzare a turno come training e validation. Per la sperimentazione si è scelto il parametro $k = 10$. Cosi facendo ad ogni passo, la 10-esima parte del dataset forma il validation, e sulla rimanente viene addestrato l'algoritmo. Questa tecnica assicura che tutto il training set (a fasi alterne) venga sfruttato per generare il modello e per fare validazione.

4.1 Metodo

Ricordiamoci quello che vogliamo dimostrare, cioè il rischio di un classificatore boostato è inferiore rispetto a quello dei singoli weak learners. Un algoritmo di classificazione parametrico come il percettrone può avere un errore di bias molto alto, mentre i classificatori ad albero subiscono il fenomeno dell'overfitting quando il numero di nodi cresce senza controllo.

4.1.1 Alberi di decisione

Per prima cosa vediamo empiricamente come è fatto l'overfitting nella famiglia di *predittori ad albero*. I parametri del dataset sono gli stessi specificati nella sezione 4. Abbiamo messo in relazione l'errore di classificazione rispetto al numero di nodi (o profondità) dell'abero. Questo

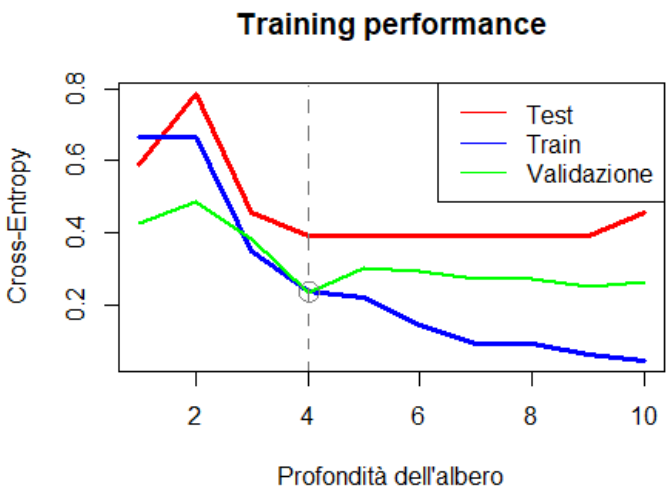

Fig. 2. Analisi delle performance - Alberi di decisione

risultato (Fig.2) mostra un training error (linea blu) in discesa quanto più addestriamo il nostro modello. Intuitivamente è quello che ci aspettiamo poichè se l'algoritmo si addestra molto sui dati, l'errore che commetterà nel classificare le medesime istanze sarà piccolo. All'aumentare della profondità dell'albero (asse x) l'errore diminuisce. Sull'asse delle ordinate abbiamo usato la Cross-Entropy come indice dell'errore. Calcolata come:

$$H(S,q) = -\sum_{i=1}^{N} \frac{1}{N} \log_2 q(x_i)$$

dove N è la taglia del training set e q la misura di probabilità. Il test error $\tilde{er}$ che in ultima analisi è quello a cui siamo interessati, diminuisce parallelamente all'errore di training nella prima parte, per poi aumentare verso la fine. Il suo valor minimo non è quindi in corrispondenza dell' $\arg\min_h \hat{er}(h)$ ma del minimo dell'errore di cross validazione cv-*er* (linea verde). Oltre questo valore siamo in overfitting. Significa che addestrando troppo il modello e facendo crescere l'albero a dismisura, il risultato che otteniamo sarà peggiore. Questo è certificato anche dalla formula del maggiorante sul rischio

$$er(\hat{h}) \leq \arg\min_{h \in H}(\hat{er}(h) + \sqrt{\frac{1}{m} O(N \log d)})$$

che come sappiamo, penalizza molto quando il numero di nodi N cresce senza controllo. Quindi

se scegliamo opportunamente l'attributo su cui andare a fare lo split, dopo soli 4 livelli di profondità il modello ci garantisce di minimizzare il rischio.
La tecnica scelta in questo test per l'indviduazione dell'attributo da cui partire per far crescere l'albero è l'*Information Gain*. Di fatto rappresenta la riduzione attesa di entropia conseguente al partizionamento degli esempi del training set. Per calcolarla è sufficiente implementare in codice R la seguente formula:

$$G(S,t) = E(S) - \sum_i \frac{|S_i|}{|S|} E(S_i)$$

Dove t è il test usato per lo split, $E(S)$ rappresenta l'entropia calcolata come $-p\log_2 p - n\log_2 n$; p ed n rappresentano in questo caso il numero di esempi positivi e negativi. Il criterio basato sull'Information Gain sceglie il test t che massimizza il guadagno $G(S,t)$. Riepilogando se la scelta del test non è quella ottimale, l'albero generato non sarà il più corto possibile e pagheremo qualcosa nella formula vista prima del maggiorante sul rischio. Questo significa che la curva di figura 2 approssimerà meno bene la funzione e^{-x}.
Si riportano di seguito gli indici di valutazione per alberi di decisione con 4 livelli di profondità.

ALBERI DI DECISIONE
$Precision = 0.88$
$Recall = 0.92$
$F - measure = 0.89$

4.1.2 Decision Stumps

Ci chiediamo, come varia l'errore quando l'attributo scelto non è quello ottimo? Abbiamo implementato il *Decision Stump* il cui comportamento è associabile a quello di un albero di decisione ad un solo livello di profondità. Usiamo una funzione randomica per la scelta del test e disinteressiamoci per un attimo del guadagno $G(S,t)$. Dal grafico di figura 3 possiamo vedere che lo split sul sesto attributo è l'unico in grado di garantire la rapida discesa del rischio $\tilde{er}(h_{stump})$. Il test su questo attributo infatti, è da preferirsi rispetto agli altri e ci garantisce di classificare nel miglior modo possibile le istanze. Non è un caso infatti che questo coincida con l'indice di guadagno più alto (la barra rossa nel grafico). Questi classificatori molto semplici non presentano overfitting e non hanno bisogno quindi di ulteriore tuning.
Riportiamo di seguito gli indici di valutazione delle performance con i decision stumps.

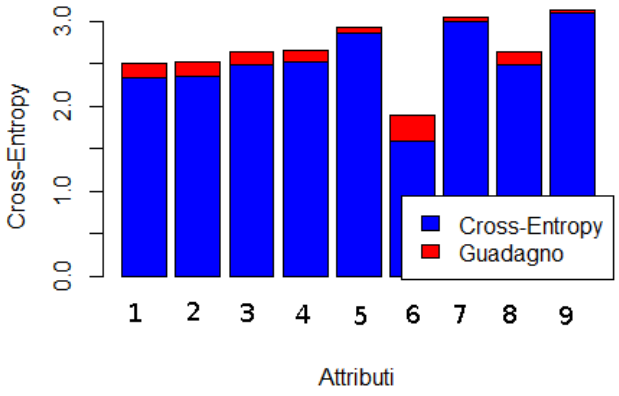

Fig. 3. Scelta del miglior attributo per il test

DECISION STUMPS
$Precision = 0.77$
$Recall = 0.95$
$F - measure = 0.85$

4.1.3 AdaBoost con Decision Stumps

Vogliamo ora dimostrare che combinando diversi decision stumps con tecniche di ensemble learning l'errore decresce e riusciamo a raggiungere migliori performance. I test condotti confrontando training e test error, mostrano che all'aumentare del numero di membri del comitato T, il training error decresce esponenzialmente come $e^{-2T\gamma^2}$. Intuitivamente significa che non sono necessari un gran numero di classificatori base per minimizzare l'errore. In figura 4 è riportato il risultato da cui si evince chiaramente che dopo soli 6 classificatori base il rischio empirico raggiunge il suo minimo. Questo risultato ci rallegra ed è in linea con la formula teorica che indica

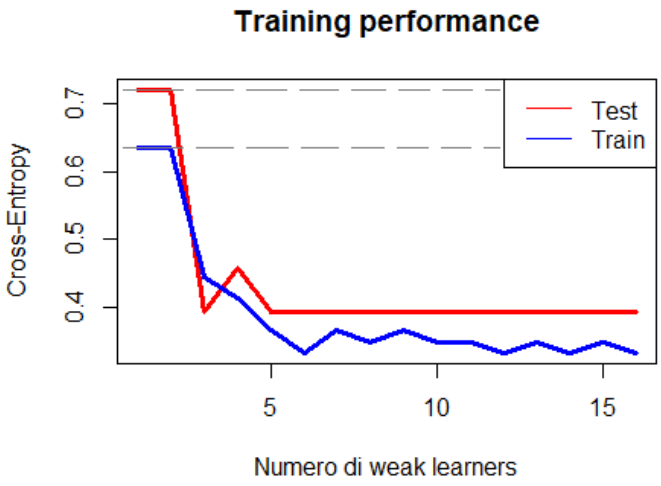

Fig. 4. Analisi delle performance - Boosting con Decision Stumps

che è sufficiente prendere T weak learners, con

$$T > \frac{\ln(m)}{2\gamma^2} \simeq O(\ln m)$$

per minimizzare l'errore di training. Qui m rappresenta la taglia del training set S. Con il nostro training set (formato da 578 esempi), il $ln(m)$ è pari a 6.4, e coincide con il risultato ottenuto durante i test.
Dai grafici di figura 2 e 4 è evidente che il training error $\hat{er}(h)$ è sempre inferiore rispetto al test error $\tilde{er}(h)$. Questo conferma il fatto che è possibile considerare il rischio empirico come un lower bound per stimare le performance teoriche di algoritmo di apprendimento.
Notiamo anche che in questo caso l'overfitting non penalizza i classificatori una volta boostati.

4.1.4 AdaBoost con Alberi di Decisione

Prendiamo alberi di decisione con 4 livelli di profondità e boostiamoli. Infatti per quanto detto nella sezione 4.1.1, alberi con questa profondità sono sufficienti per minimizzare il training error senza rischiare di cadere nella trappola dell'overfitting. Possiamo ottenere delle performance migliori rispetto a quelle dei singoli weak learners? Per prima cosa guardiamo la *Confusion Matrix* relativa agli esempi di test; quelli cioè che l'algoritmo non ha mai visto per allenarsi.

Confusion Matrix

		p	**n**	**totale**
Valore attuale	**p'**	30 29.4%	2 1.9%	32
	n'	1 0.9%	69 67.6%	70
	totale	31	71	

Su un totale di 102 esempi l'algortmo sbaglia a classificarne soltanto 3. Di seguito riportiamo gli indici di precision & recall che confermano un miglioramento nel classificare le istanze di test.

ADABOOST
$Precision = 0.96$
$Recall = 0.94$
$F - measure = 0.95$

Abbiamo maggior evidenza di questo fatto, guardando il grafico di figura 5 che calcola la coppia recall/precision per ogni posizione del training set. In poche parole, rappresenta la stima di precisione ad ogni livello standard di recall. La curva più vicina al punto (1,1) indica la migliore prestazione.
Dal grafico viene confermato quanto finora specificato a livello teorico. I predittori decision stumps sono quelli che hanno le peggiori performance. Infatti la funzione (monotona decrescente) tra tutte è quella che si discosta maggiormente dai punti (1,1). Al contrario gli alberi di decisione con AdaBoost hanno un andamento che approssima meglio la funzione $y = 1$. Nel mezzo i risultati ottenuti con alberi di decisione quando non vengono usati insieme ad algoritmi di boosting.

4.1.5 Percettrone

Spostiamo ora la nostra attenzione su un'altro algortimo di classificazione il Percettrone. Sappiamo che è parametrico e genera classificatori lineari. Usiamolo per classificare le nostre istanze e analizziamo come si comporta su dati generati

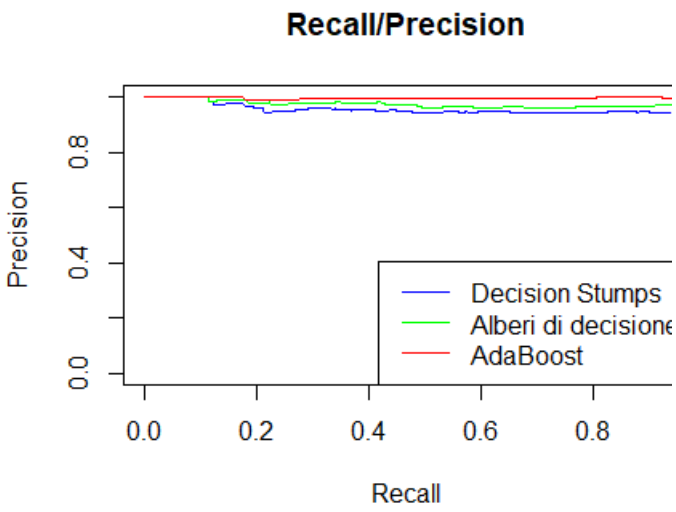

Fig. 5. Algoritmi a confronto - Recall/Precision

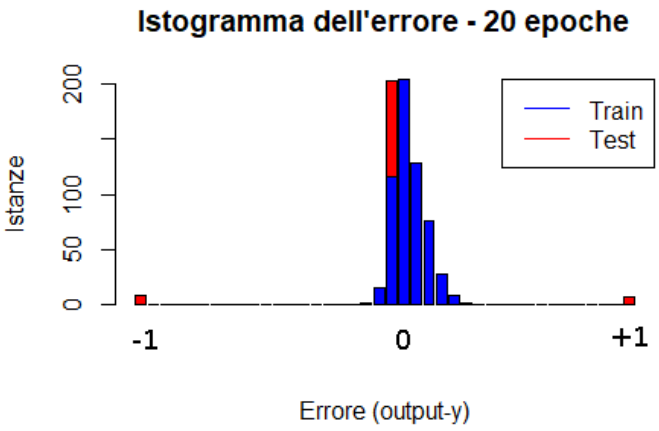

Fig. 6. Istogramma dell'errore di train e test

Table 1. Errori con un test set di 102 esempi

Esempio	Epoche	Errori
1	10	31
2	50	12
3	100	10
4	300	17
5	500	15
6	750	9
7	1000	13
8	2000	12
9	5000	13
10	10000	12

dalla stessa distribuzione. Se il bayesiano ottimo avesse una struttura rigida (ad esempio se fosse un iperpiano), saremmo sicuri che in un numero finito di passi, il nostro modello convergerebbe al miglior modello f^*. Significa che un semplice iperpiano sarebbe sufficiente per classificare bene tutti gli esempi.

Diviso come al solito il dataset in training e test, ad ogni epoca abbiamo rimescolato le istanze. Questa non è una procedura obbligatoria, ma può essere una buona prassi farlo. Ad ogni ciclo di iterazione ogni istanza ha la stessa probabilità di essere ripescata e contribuire alla generazione del modello. Durante la fase di addestramento il nostro iperpiano non converge mai, segno del fatto che il training set non è linearmente separabile e quindi indipendentemente dal numero di epoche, non raggiungiamo mai la convergenza.

Cerchiamo quindi di capire qualcosa di più sulla distribuzione degli errori e guardiamo come questi sono legati alle istanze. Abbiamo plottato sul grafico di figura 6, l'errore medio che commette l'iperpiano nel classificare ogni singola istanza durante le n epoche. In blu sono rappresentate le istanze di training (85% del totale) ed in rosso quelle di test. La curva assume la forma di una Gaussiana centrata nel punto zero. Indice che la distanza tra etichetta reale Y ed etichetta predetta $\hat{y}$ per tutti gli esempi ed in media, si avvicina a zero. Aumentando il numero di epoche la curva approssimerà sempre meglio la funzione *Delta di Dirac* poichè l'errore commesso in un'epoca diventerà sempre meno rilevante.

In questo specifico esempio gli errori di test sono 17 (barra rossa). Siamo lontani dalle performance di 3 errori ottenute boostando gli alberi di decisione. In realtà abbiamo notato che aumentando il numero di epoche, il numero di errori non sembra diminuire in modo significativo. La tabella 1 riporta il numero di errori commessi con un test set di 102 esempi in base ad un diverso numero di

Table 2. Errori di test con diverso numero di weak learners

weak learners	Errori
2	4
4	3
6	3
8	3
10	8

epoche. Il modello generato allenando l'algoritmo con soli 10 epoche risulta in effetti quello peggiore. Le performance con 10.000 epoche però non risultano migliori rispetto a quelle con 50 epoche. Intuitivamente questo potrebbe essere dovuto al fatto che le nostre istanze non sono ben centrate intorno all'origine degli assi.

4.1.6 AdaBoost con Percettrone

Come ultima analisi, abbiamo boostato il percettrone per capire se riusciavamo a migliorare i risultati precedenti. Utilizzando per classificare iperpiani con un valore di soglia diverso da zero, siamo riusciti a vedere qualche miglioramento. I miglioramenti sono legati puramente a questo aspetto e non al numero di classificatori base adoperati. Infatti dai risultati sul boosting non riusciamo ad esibire una correlazione tra l'aumento del numero di classificatori ed una diminuzione costante del test error. Il sospetto è che il training set non sia centrato intorno all'origine e combinando classificatori non si riesca a migliorare la situazione.
In tabella 2 riportiamo un esempio del numero di errori di test con un diverso numero di classificatori base. Come si può notare rispetto alla tabella 1, il numero di errori decresce molto, ma la cosa non sembra essere correlata in alcun modo con il numero di weak learner estratti ed usati dall'algoritmo.

5 Sommario e conclusioni

Dai test risulta chiaro quanto affermato dal teorema del *"no free lunch"*. Non esiste un unico algoritmo che risolve qualsiasi problema meglio di altri, ma è necessario valutare di volta in volta l'algoritmo migliore. Infatti il percettrone su training set linearmente separabili è in grado di dare ottime performance. Risulta chiaro in questi test, che il boosting con alberi di decisione sembra poter essere la scelta più azzecata per classificare meglio le istanze.
In particolare abbiamo dimostrato su questo dataset, che boostando predittori semplici ed economici come i decision stumps, riusciamo ad ottenere buone prestazioni. Il confronto con il percettrone ha invece messo in risalto alcuni limiti dei classificatori lineari quando il dataset non è linearmente separabile. Un approccio alternativo ci porta ad adoperare tecniche di trasformazione dei predittori parametrici in non parametrici. Si può pensare per esempio, di mappare le features in spazi a molte più dimensioni ed apprendere iperpiani direttamente in questi spazi (es. Kernel Gaussiani), ma ciò esula dagli obiettivi di questo paper. Proprio quest'ultimo aspetto potrebbe riservare uno scenario di sviluppi futuri al lavoro di analisi fin qui fatto.

6 Appendice

Tutti i files del progetto sono contenuti all'interno della cartella denominata: *"Progetto_MSA_CristianLepore"*. La cartella contiene il dataset utilizzato e tre sotto-directory.

Decision_stumps: al suo interno si trovano i files utilizzati per implementare il decision stump separati tra la versione con boosting e senza.
Decision_tree: oltre ai file del codice troviamo anche gli script per la costruzione dei grafici di Recall/precision ed Information gain.
Percettrone: contiene gli script dell'implementazione dell'algoritmo e generazione dei grafici.

Una volta importato il dataset i file possono essere compilati nell'ambiente RStudio senza ulteriori modifiche.

7 Riferimenti

[1] Dispense del corso di Metodi Statistici per l'Apprendimento. Prof.re Nicolò Cesa-Bianchi Pagina web: cesa-*bianchi.di.unimi.it/MSA/*
[2] UCI Machine Learning Repository. Pagina web: *archive.ics.uci.edu/ml/index.php*
[3] Software adoperati, RStudio. Pagina web: *rstudio.com/*