Classificazione testuale mediante Perceptron

Samuele Meta

February 9, 2018

1 Introduzione

Inventato nel 1957 da Frank Rosenblatt, il Perceptron è una delle prime reti neurali artificiali ad essere stata prodotta. Tale algoritmo viene utilizzato nell'ambito dell'apprendimento supervisionato, ovvero basato su un insieme di dati pre-etichettati, per allenare classificatori binari e separare, mediante un piano, gli esempi delle due classi in esame. Fornita una strategia implementativa del suddetto algoritmo e delle sue varianti nel linguaggio *Python*, lo scopo di questo progetto è quello di sperimentarne l'utilizzo sul dataset *20 Newsgroup*.

2 Il Perceptron

L'algoritmo originario prevede, per la fase di *training*, di ricevere in ingresso un insieme di dati accompagnati da un'etichetta che specifica la loro classe di appartenenza. Viene quindi inizializzato un vettore $\vec{w}$, rappresentante il piano che agisce da classificatore, e b, il *bias*, che indica la distanza del piano dall'origine. Per ciascun esempio $\vec{x}$ sarà computata l'operazione:

$$\hat{y} = \vec{x} \cdot \vec{w} + b$$

Nel caso in cui il segno di $\hat{y}$ non sia concorde con quello dell'etichetta y associata al dato esaminato, sarà necessario aggiornare i coefficienti di $\vec{w}$ e b per tener conto della nuove informazioni apportate dall'ultimo dato. Questo avviene mediante le operazioni:

$$\vec{w} = \vec{w} + y \cdot \vec{x}$$

$$b = b + y$$

Nell'altra eventualità si potrà invece passare all'input successivo poiché la classificazione è risultata esatta secondo quel piano.

2.1 Perceptron votato

Nonostante la versione standard fornisca un buon risultato nella sua semplicità, questa presenta alcune criticità non indifferenti. Supponiamo infatti di voler classificare la relativamente semplice funzione XOR. Appare subito evidente l'impossibilità di tracciare un piano che divida gli esempi positivi da quelli negativi senza commettere alcun errore. Ne consegue che l'algoritmo, non essendo i dati linearmente separabili, continuerà a generare ogni volta un piano diverso e, quello finale, sarà determinato casualmente dal momento in cui avverrà l'arresto a seguito di un certo numero di iterazioni. Supponiamo poi di eseguire il *training* su un dataset e di ottenere, dopo alcune iterazioni, un soddisfacente classificatore che predice correttamente i successivi 5000 dati sottoposti. Nel caso l'ultimo dato venisse però classificato erroneamente, è previsto che il piano debba essere aggiornato nonostante la sua accuratezza precedente. Per limitare queste situazioni, ogniqualvolta un piano debba cambiare, questo verrà salvato associandogli il numero c di classificazioni consecutive corrette. Così facendo, in fase di *testing*, sarà possibile determinare il segno di un esempio andando a pesare il contributo di ciascun piano:

$$s = \sum_{i=1}^{k} c_i \cdot sign(\vec{w} \cdot \vec{x})$$

3 Dal testo ai dati in ingresso

Il dataset *20 Newsgroup* è costituito da documenti sotto forma di email trattanti diversi argomenti e, in base a questi, classificati in categorie. Poiché il Perceptron richiede in ingresso un vettore numerico è innanzitutto necessario riuscire a rappresentare in tale forma l'informazione.

3.1 Rielaborare il testo

Nell'eseguire il caricamento del *dataset*, dopo aver scelto le due categorie su cui lavorare, è possibile agire sui parametri della funzione. A tal proposito, si rivela particolarmente importante aggiungere la rimozione di *header*, *footer* e *quotes* dai testi dei messaggi. Infatti, come suggerito dalla documentazione della libreria *Scikit-learn*, questi elementi potrebbero influenzare negativamente l'apprendimento dell'algoritmo perché poco attinenti al problema della classificazione. Nonostante da questa scelta derivi un notevole calo dell'accuratezza in fase di *testing*, tale coefficiente sarà più realistico.

3.2 Quantificare il testo

Il primo passo è quello di generare, tramite *CountVectorizer*, un dizionario di tutte le parole contenute nei documenti trattati. Dal momento che alcuni termini ricorrono più frequentemente di altri nell'uso di una lingua, si è ritenuto opportuno escludere articoli, preposizioni e, più in generale, le *stop words* che non contribuiscono in modo significativo a nessuna classe. Ogni vettore rappresentante l'input avrà lunghezza pari a quella del vocabolario e di conseguenza sarà fortemente sparso, con pochi valori diversi da zero. Se in prima battuta si potrebbe pensare di associare ad ogni parola nel dizionario il numero di volte che compare all'interno di un documento, ci si può facilmente rendere conto che così testi di diversa lunghezza sarebbero trattati differentemente. Per far fronte a questa problematica è stata applicata una trasformazione TF-IDF[1] fornita da *TfidfTransformer*.

4 Esperimenti

Dal momento che studi sul dataset[2] hanno evidenziato un buon bilanciamento dei vari dati, non è stato necessario eseguire la *Cross Validation*. Da diverse esecuzioni dell'algoritmo si è invece potuto osservare che un diverso ordinamento dei dati su cui viene eseguito il *training* influisce sulla cronologia dei vettori e sui loro pesi, determinando accuratezze diverse. Si è ritenuto inoltre interessante tenere traccia del consumo di memoria delle due versioni e il numero di classificatori utilizzati da quella votata.

4.1 Rec.Autos vs Rec.Motorcycles

		Standard			Votato	
		Auto	**Moto**	**Total**	**Auto**	**Moto**
Reali	**Auto**	392	4	**396**	290	122
	Moto	213	185	**398**	30	352
	Total	605	189	**794**	320	474

In caso di perceptron votato si ottiene un'accuratezza dell'80.85% a fronte di un consumo di memoria di 351.1 MiB. Nel caso standard invece risulta

[1] Dall'inglese *Term Frequency - Invers Document Frequency*, permette di evidenziare l'importanza di una parola all'interno di un testo. Questo viene fatto non solo dividendo il conteggio per la lunghezza del documento, ma anche abbassando il peso delle parole più ricorrenti nei vari testi.

[2] Consultabili su cn-static.udacity.com/mlnd/Capstone_Poject_Sample01.pdf

72.67% di accuratezza per 246.1 MiB di memoria. Cambiando poi il seme secondo il quale viene permutato l'ordine dei dati in ingresso da 36 a 18:

		Standard			Votato	
		Auto	**Moto**	**Total**	**Auto**	**Moto**
Reali	**Auto**	280	131	**410**	301	110
	Moto	25	358	**384**	34	349
	Total	305	489	**794**	335	459

Possiamo osservare come in questo caso l'accuratezza del caso standard sia del 80.35%, mentre per il perceptron votato rimanga simile (81.86%).

4.2 Sci.Crypt vs Sci.Electronics

		Standard			Votato	
		Auto	**Moto**	**Total**	**Auto**	**Moto**
Reali	**Auto**	204	192	**410**	349	58
	Moto	2	391	**384**	76	349
	Total	206	583	**789**	425	407

Il perceptron standard realizza 75.41% di accuratezza e 314.6 MiB in memoria, mentre quello votato 83.01% di accuratezza e 414.6 MiB occupati.

4.3 Comp.Graphics vs Comp.Sys.Mac.Hardware

		Standard			Votato	
		Auto	**Moto**	**Total**	**Auto**	**Moto**
Reali	**Auto**	359	44	**410**	352	51
	Moto	49	322	**384**	41	330
	Total	408	366	**774**	393	381

Il perceptron standard realizza 87.98% di accuratezza e 266.1 MiB in memoria, mentre quello votato 88.11% di accuratezza e 361.7 MiB occupati.

5 Conclusioni

Come è logico che sia, non è possibile sapere a priori se la versione votata comporterà un vantaggio effettivo rispetto a quella standard. Se di sicuro impiegherà più tempo e memoria, a causa dell'ordine dei dati potrebbe anche verificarsi che i due approcci convergano alla stessa soluzione rendendo la votazione superflua. Resta comunque un difetto trascurabile alla luce del fatto che garantisce sempre un certo livello di accuratezza, cosa che lo rende preferibile all'approccio standard reo talvolta di scendere anche al 60%.