Università degli Studi di Modena e Reggio Emilia

Dipartimento di Ingegneria "Enzo Ferrari"
Corso di Laurea in Ingegneria Informatica

Valutazione di tecniche di Machine Learning per l'Entity Resolution

Relatore:
Prof.ssa Sonia Bergamaschi

Candidato:
Stefano Gavioli

Anno Accademico 2017/2018

Indice

Introduzione 1

1 Entity Resolution 2

2 Machine Learning 5

2.1 Artificial Neural Network . . . 8

2.1.1 Perceptron . . . 8

2.1.2 Strutture . . . 16

2.1.2.1 Feed-forward Neural Network . . . 16

2.1.2.2 Recurrent Neural Network . . . 19

2.2 Support Vector Machine . . . 22

3 Algoritmo 24

3.1 Deeper Model . . . 24

3.1.1 Average . . . 24

3.1.2 LSTM . . . 25

3.1.3 Bi-LSTM . . . 27

3.2 Word Representation . . . 29

3.2.1 Pre-made embeddings . . . 29

3.2.1.1 GloVe . . . 30

4 Test 31

4.1 Fodor's Zagat . . . 31

4.2 DBLP ACM . . . 35

4.3 Amazon Google . . . 41

4.4 Riduzione del training set . . . 45

4.5 Considerazioni finali . . . 48

Riferimenti Bibliografici 52

A Ambiente 53

A.1 Specifiche . . . 53

A.2 Python . . . 53

A.3 Keras 53

B Codice **54**

B.1 Struttura 54

B.2 Input 54

B.3 Data Preparation 55

B.4 Modello 56

Elenco delle figure

1 Caption . . . 2
2 Precision e Recall . . . 6
3 F_1 3d plot . . . 7
4 Perceptron . . . 8
5 $E(\mathbf{w})$ con una sola feature . . . 9
6 Gradient Descent effetto ping-pong . . . 10
7 Gradient descent con eta molto piccolo . . . 11
8 Gradient Descent, $E(\mathbf{w})$ in relazione agli steps . . . 12
9 Sigmoide $\sigma(a)$ e tangente iperbolica $tanh(a)$ . . . 15
10 Strato di percettroni . . . 16
11 Percettrone multistrato, MLP . . . 17
12 Recurrent Neural Network . . . 20
13 Long Short-Term Memory unit . . . 21
14 Support Vector machine . . . 22
15 Average single layer neural network . . . 25
16 Doppia LSTM . . . 26
17 Singola LSTM . . . 27
18 Doppia Bi-LSTM . . . 28
19 Singola Bi-LSTM . . . 28
20 Fodor's Zagat Dataset . . . 33
21 Fodor's Zagat F_1 . . . 34
22 Fodor's Zagat tempo di training . . . 35
23 DBLP ACM Dataset . . . 37
24 DBLP ACM F_1-score . . . 39
25 DBLP ACM tempo di training . . . 40
27 Amazon Google F_1-score . . . 43
26 Amazon Google Dataset . . . 44
28 Amazon Google tempo di training . . . 45
29 Confronto sul tempo di training variando la percentuale di training set 46
30 Confronto sul F_1 score variando la percentuale di training set . . . 47
31 Confronto sul F_1 score rispetto al test set variando la percentuale di training set . . . 48
32 Rete Neurale milgiore . . . 49

33 Struttura del codice 54
34 Espansione della struttura 55
35 Interfaccia tra i livelli Input e Data Preparation 55
36 Livello Data Preparation 56

Introduzione

L'Entity Resolution, conosciuto anche con altri nomi, per esempio Entity Matching, non è un problema di recente insorgenza, bensì una questione affrontata per più di 70 anni[1] e viene introdotto in modo più approfondito nella sezione 1.

Anche le tecniche di Machine Learning non sono moderne considerando che affondano le loro radici fino agli anni '40-'50. Ciò nonostante tali tecniche sono diventate recentemente molto utilizzate, soprattutto in relazione agli sviluppi tecnologici che hanno permesso un aumento delle capacità computazionali, permettendo l'esecuzione di questi in tempi inferiori e quindi più accessibili.

L'applicazione di metodi di Machine Learning, illustrati nella sezione 2, al problema di Entity Resolution è lo stato dell'arte. Questo elaborato considera i vari metodi di machine learning proposti nel paper "Distributed Representations of Tuples for Entity Resolution" da Ebraheem et al [1], illustrato nella sezione 3, per valutarli rispetto a diversi dataset di benchmark largamente utilizzati[2, 3], i cui risultati sono illustrati nella sezione 4, allo scopo di identificare infine quali tra questi fornisca le prestazioni migliori.

Le appendici A e B contengono rispettivamente una breve descrizione dell'ambiente in cui sono stati effettuati i test e la struttura del codice utilizzato.

1 Entity Resolution

Per *Entity Resolution*, anche chiamato *Entity Matching* o *Record Linkage*, si intende quell'attività che identifica due istanze di dati facenti riferimento alla stessa entità reale. Per esempio il secondo record nella Tabella 1, ed il secondo record nella Tabella 2 anche se non identici sono legati alla stessa entità reale: Mario Rossi. Questa definizione è molto lasca e la funzione può dunque presentarsi in diverse forme a seconda del formato dei dati.

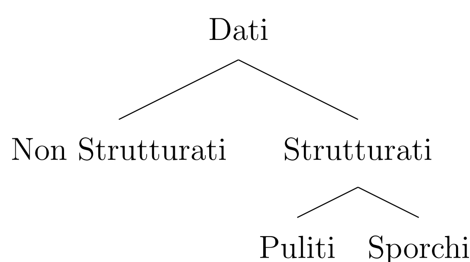


Figura 1: Caption

STRUTTURATI: I dati sono così definiti in quanto sono un insieme di record o di tuple, più precisamente un'istanza di relazione, come per esempio in Tabella 1.

Nome	Cognome	Matricola
Stefano	Gavioli	103012
Marco	Rossi	123456

Tabella 1: Dati Strutturati

Nome	Cognome	Matricola
Giovanni	Rossi	654321
Marco Rossi		123456

Tabella 2: Dati Strutturati Sporchi

Strutturati Sporchi: Sono dati strutturati, un'istanza di relazione, ma in diversi record sono presenti errori; in particolare in alcuni record valori di un attributo possono non essere presenti ed apparire in un valore di un altro attributo come in Tabella 2.

Non Strutturati: Sono dati che non fanno riferimento a nessuna relazione, quindi, nel caso di descrizioni testuali, non sono altro che un semplice corpo testo, per esempio una descrizione.

Un altro fattore rilevante che differenzia algoritmi di *Entity Resolution* è l'origine dei dati. Nel caso di due o più sorgenti[1] lo schema di relazione può differire da sorgente a sorgente. Si definiscono allineati due schemi che sono composti dagli stessi attributi. Nel caso questi non fossero allineati si possono usare tecniche di allineamento, come per esempio estrapolare un sottoinsieme di attributi che contengono informazioni sufficientemente simili.

Nel caso di una singola sorgente gli schemi sono necessariamente allineati. In questo caso particolare l'algoritmo di Entity Resolution è più comunemente noto come Deduplication. Nel nostro caso verranno usati esclusivamente dati strutturati, sia puliti che sporchi, da due sorgenti i cui schemi, composti da quattro attributi, sono allineati.

Volendo dare una formulazione più formale al problema: siano $A_j, \quad j = 1, \ldots, m$ le istanze delle m sorgenti, chiamiamo U l'insieme unione di tutte le m istanze:

$$U = \bigcup_{j=1}^{m} A_j$$

e chiamiamo D il prodotto cartesiano di U con se stesso

$$D = U \times U = U^2$$

D conterrà tutte le coppie $(i, j) \quad i, j \in U$
Possiamo ora definire una funzione stato $s : D \to \{Vero, Falso\}$, se $i \in U$ e $j \in U$ fanno riferimento alla stessa entità reale allora $s(i, j) = Vero$, altrimenti $s(i, j) = Falso$. Ne segue che naturalmente

$$s(i, i) = Vero \quad \forall i \in U$$

ma anche che

$$s(i, j) = s(j, i)$$

[1]Per sorgente si intende una singola relazione e i dati relativi ad essa sono la sua istanza

Considerando che ogni coppia $(i, j) \in D$ dev'essere opportunamente visionata per determinarne $s(i, j)$ valutare l'intero insieme D, che ha cardinalità $\#D = \#U^2$ può essere inutilmente computazionalmente dispendioso e si può quindi considerare un insieme $D' \subset D$, che non contenga le coppie invertite; ovvero se $(i, j) \in D'$, allora $(j, i) \notin D'$. Questo riduce la quantità di confronti da effettuare a $\#D' = \begin{pmatrix} \#U \\ 2 \end{pmatrix} < \#D = \#U^2$
Ciò può comunque risultare molto dispendioso, per esempio nel caso di $\#U \approx 10^5$, $\#D' \approx 5 \times 10^9 < 10^{10} = \#U^2$

Ci sono ulteriori tecniche per diminuire il numero di confronti da effettuare. Queste tecniche, dette di *blocking*, considerano un sottoinsieme U' dell'insieme U. U' è costruito cercando di minimizzare i *mismatch* ovvero tutte le coppie $(i, j) \in U - U'$ tali che $s(i, j) = Vero$, estrapolando informazioni: dallo schema, esclusivamente dai dati (quindi ignorando lo schema), od entrambi[4].

2 Machine Learning

Con Machine Learning si intendono tutti gli algoritmi che imparano dall'esperienza. Più precisamente, dandone una definizione più formale:

Si dice che un programma impari dall'esperienza E rispetto a una qualche classe di attività T e misura di prestazione P, se le sue prestazioni alle attività in T, misurate con P, migliorano con l'esperienza E [5].

Nel caso dell'Entity Resolution sono:

ATTIVITÀ T) Riconoscere una coppia di record i cui elementi fanno riferimento alla stessa entità reale.

ESPERIENZA E) Coppie di record di cui lo stato è noto.

PRESTAZIONE P) Per misurare le prestazioni si usano due metriche: *precision* e *recall*[6].

PRECISION) Chiamata anche *confidence*, misura la percentuale dei veri correttamente classificati rispetto a tutti i classificati come veri.

$$precision = \frac{\text{true positives}}{\text{true positives} + \text{true negatives}}$$

RECALL) Chiamata anche *sensitivity*, misura la proporzione dei predetti veri correttamente classificati rispetto a tutti i veri.

$$precision = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}$$

Per esempio supponendo di avere 100 coppie di cui 10 sono vere e 90 false, e supponendo che un determinato algoritmo identifichi come vere 5 coppie delle quali 2 sono realmente vere e 3 sono in realtà false. Allora la $precision = \frac{2}{5} = 40\%$ e la $recall = \frac{2}{10} = 20\%$.

Vi è una ulteriore difficoltà nel cercare di comprendere quali coppie di precision e recall sono migliori rispetto ad altre. Ad esempio $(90.6\%, 92.2\%)$ e $(93.2\%, 89.9\%)$ sono coppie i cui valori sono molto vicini nei valori e quindi determinare la coppia migliore può essere difficile.

F_1 SCORE) A tale scopo se ne calcola la media armonica che viene chiamata F_1 score[7]. Si utilizza la media armonica in quanto sia precision che recall hanno i *true positives* al numeratore e quindi per avere un confronto omogeneo si confrontano i reciproci.

$$\begin{aligned} F_1 &= \frac{2}{\frac{1}{precision} + \frac{1}{recall}} \\ &= \frac{2}{\frac{precision+recall}{precision \cdot recall}} \\ &= 2\frac{precision \cdot recall}{precision + recall} \end{aligned}$$

Questo valore ci permette di confrontare diverse coppie di precision e recall. Prendendo come esempio quello precedente otteniamo rispettivamente due valori di F_1: 91.39% e 91.52% e quindi con una differenza di 0.13% possiamo affermare che il secondo risultato è lievemente migliore.

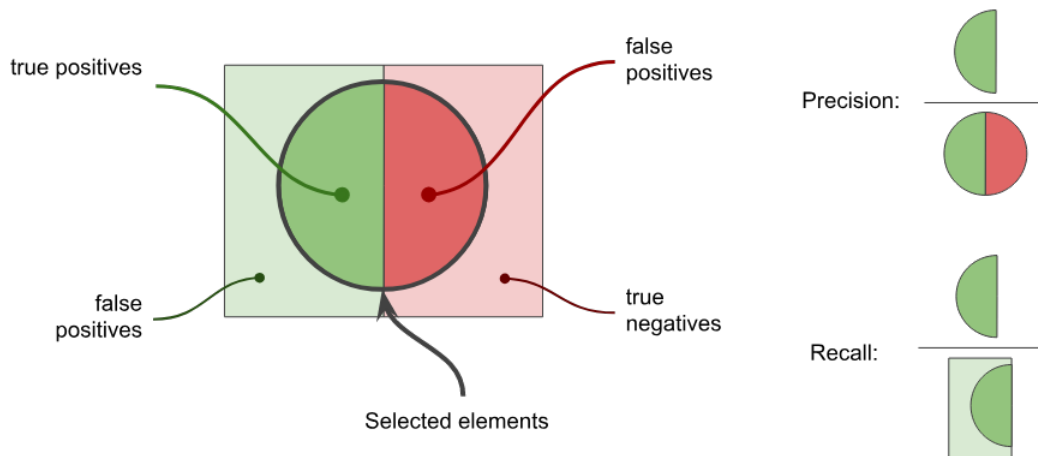


Figura 2: Precision e Recall. *Selected Elements* rappresenta gli elementi indicati come veri dall'algoritmo in considerazione

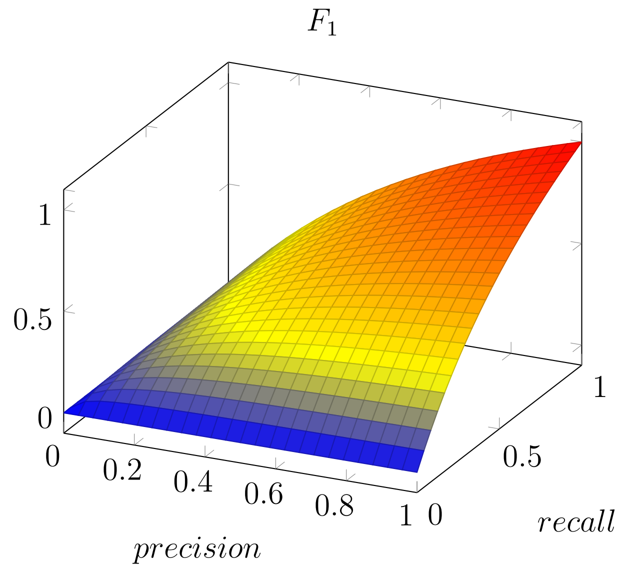


Figura 3: F_1 al variare di *precision* e *recall*

Come si può notare nell'esperienza E i dati a disposizione dell'algoritmo contengono anche informazioni sul loro stato. Questo significa che i vari algoritmi valutati usano tecniche di *Supervised Learning*.

Esistono anche altre metodologie di *learning*: *Unsupervised Learning* in cui l'algoritmo non ha a disposizione le suddette informazioni e quindi deve autonomamente identificare il modello e comprendere come i dati siano organizzati; *Semi-Supervised Learning* in cui si hanno sia dati i cui valori obiettivo sono stati indicati (*labeled data*), che dati in cui non sono disponibili(*unlabeled data*); *Reinforcement Learning*, in questo caso si cercano le azioni corrette per massimizzare una ricompensa, per esempio un algoritmo può imparare a giocare a scacchi e le ricompense da massimizzare sono le vittorie [8].

2.1 Artificial Neural Network

Le reti neurali artificiali sono così chiamate in quanto l'elemento fondamentale di queste reti, il *perceptron*, è ispirato da osservazioni fatte sulle reti neurali biologiche. Questi sono stati impiegati sia per studiare modelli di apprendimento biologici che per sviluppare modelli di apprendimento artificiale, indipendentemente che questi siano realmente rappresentativi di processi biologici [5].

Un insieme di neuroni artificiali formano una rete neurale artificiale. Come questi siano interconnessi forma la struttura di una rete neurale a seconda del quale la rete assume una differente denominazione. Infatti i neuroni all'interno della rete possono presentare connessioni cicliche formando dunque le reti neurali ricorrenti (vedi sezione 2.1.2.2), oppure nel caso non le presentino le reti *feed-forward* (vedi szione 2.1.2.1).

Esistono anche altri tipi di strutture le quali però, non essendo di interesse all'algoritmo illustrato nella sezione 3 non vengono illustrate.

2.1.1 Perceptron

Il percettrone fu inventato da Frank Rosenblatt nel 1957-1958, il quale sviluppò anche il primo neuro-computer, il *perceptron Mark I* [9].

Viene ora illustrato il modello di percettrone e le varie funzioni che sono associate.

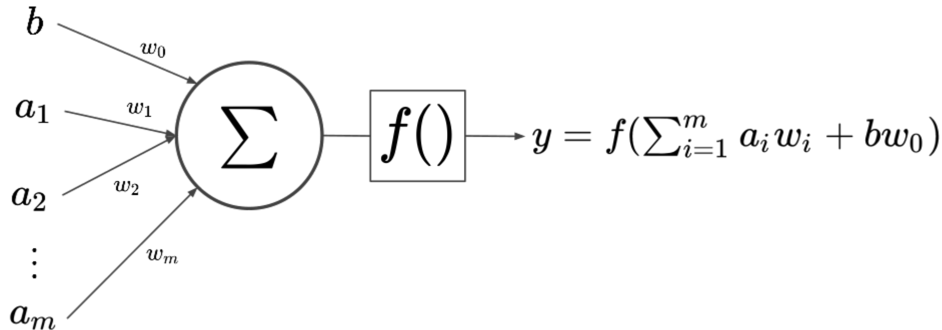


Figura 4: Modello di perceptron

Avendo a sinistra $\mathbf{a} = (a_1, a_2, \ldots, a_m)$ e $b = 1$ rispettivamente i vari ingressi del perceptron ed il *bias*, ad ogni input a_i è associato un valore reale chiamato peso $w_i \in \mathbb{R}$. L'uscita dipende sia dai pesi che dagli ingressi.

Possiamo chiamare $\mathbf{x} = (b, a_1, \ldots, a_m) \equiv (1, a_1, \ldots, a_m) \in \mathbb{R}^{m+1}$ e $\mathbf{w} = (w_0, w_1, \ldots, w_m) \in \mathbb{R}^{m+1}$. Il perceptron applica una funzione non necessariamente lineare $f : \mathbb{R}^{m+1} \to \mathbb{R}$

detta funzione di attivazione al prodotto scalare $\mathbf{x} \cdot \mathbf{w}$. L'uscita y risulta quindi:

$$\begin{aligned} y &= f\left(\sum_{i=1}^{m} a_i w_i + b w_0\right) \\ &= f(\mathbf{a} \cdot \mathbf{w} + b w_0) \\ &= f(\mathbf{x} \cdot \mathbf{w}) \end{aligned}$$

L'uscita $y(\mathbf{x}, \mathbf{w})$ è una funzione sia degli ingressi che dei pesi. Variando i pesi si ottengono risultati diversi ed è qui che avviene l'apprendimento, modificando i valori di $\mathbf{w}$ in relazione al risultato ottenuto ed al risultato desiderato mediante tecniche illustrate in seguito.

Sia $D = \{(x^{(0)}, t^{(0)}), (x^{(1)}, t^{(1)}), ..., (x^{(k)}, t^{(k)})\}$, l'insieme dei valori che saranno usati per l'apprendimento, quindi chiamato *training set*, per ognuno dei valori $\mathbf{x^{(i)}}$ esiste il valore in uscita $o^{(i)}$ e il valore desiderato $t^{(i)}$.
É quindi possibile definire una funzione errore sull'intero insieme D per esempio:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=0}^{k} (t^{(i)} - o^{(i)})^2$$

Il nostro scopo è intuitivamente minimizzare l'errore $E(\mathbf{w})$

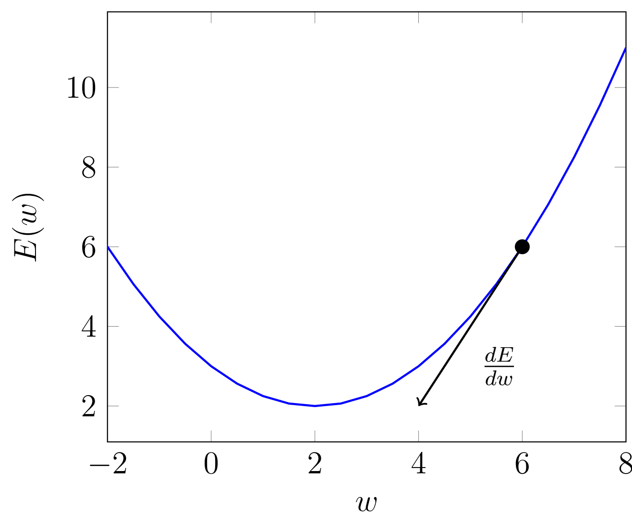


Figura 5: $E(\mathbf{w})$ considerando ingressi $x \in R$

L'algoritmo *gradient descent* inizializza casualmente i pesi $\mathbf{w}$ dopodiché procede modificandoli nella direzione della discesa maggiore lungo la curva dell'errore, finché il minimo non è stato raggiunto.

Tale direzione si ricava derivando $E(\mathbf{w})$ rispetto ad ogni componente di $\mathbf{w}$:

$$\nabla E(\mathbf{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_m}\right]$$

Tale direzione rappresenta la direzione del massimo incremento, dunque si considera il suo opposto.

Considerando l'aggiornamento dei pesi:

$$\mathbf{w} : \mathbf{w} + \Delta\mathbf{w}$$

allora

$$\Delta\mathbf{w} = -\eta\nabla E(\mathbf{w})$$

che può essere scomposta per componente:

$$w_i : w_i + \Delta w_i$$

$$\Delta w_i = -\eta\frac{\partial E}{\partial w_i}$$

$\eta \in \mathbb{R}^+$ è chiamato *learning rate* e rappresenta di quanto si proceda nella direzione indicata dal gradiente nel minimizzare i pesi. É un parametro molto importante infatti nel caso fosse troppo grande si potrebbe rischiare di ottenere un effetto ping-pong.

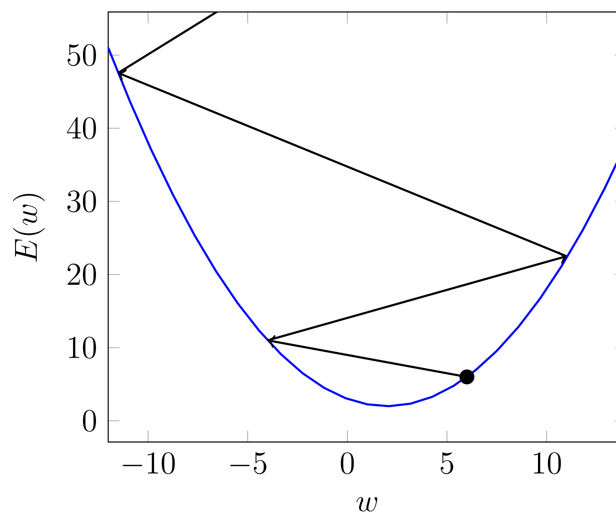


Figura 6: Effetto ping-pong

Nel caso η fosse troppo piccolo, ci vorrebbero tanti passi. In questo caso il minimo sarebbe comunque raggiunto anche se solo dopo un elevato numero di passi.

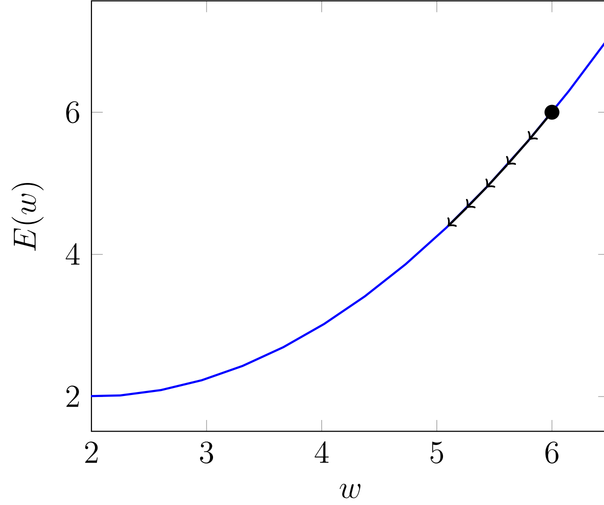


Figura 7: Nel caso η sia molto piccolo

Una applicazione abbastanza comune è diminuire η man mano che si procede con i passi [5].

Data la $E(\mathbf{w}) = \frac{1}{2}\sum_{j=0}^{k}(t^{(j)} - o^{(j)})^2$ e supponendo che la funzione di attivazione f sia la funzione identità, allora si ricava che il termine $\frac{\partial E}{\partial w_i}$:

$$\begin{aligned}
\frac{\partial E(\mathbf{w})}{\partial w_i} &= \frac{\partial}{\partial w_i}\sum_{j=0}^{k}(t^{(j)} - o^{(j)})^2 \\
&= \frac{1}{2}\sum_{j=0}^{k}\frac{\partial}{\partial w_i}(t^{(j)} - o^{(j)})^2 \\
&= \sum_{j=0}^{k}(t^{(j)} - o^{(j)})(-\frac{\partial}{\partial w_i}o^{(j)})^2 \\
&= \sum_{j=0}^{k}(t^{(j)} - o^{(j)})(-x_i^{(j)})
\end{aligned}$$

E quindi l'aggiornamento dei pesi diventa:

$$\Delta w_i = \eta\sum_{j=0}^{k}(t^{(j)} - o^{(j)})x_i^{(j)}$$

Questo aggiornamento viene effettuato finché $E(\mathbf{w})$ non ha raggiunto il minimo. Questo si osserva quando $E(\mathbf{w})$ non varia più nel tempo. Esso può non necessariamente essere il minimo globale qualora la funzione E presenti dei minimi locali.

Mostrando la relazione di $E(\mathbf{w})$ ed i passi (detti anche *steps*):

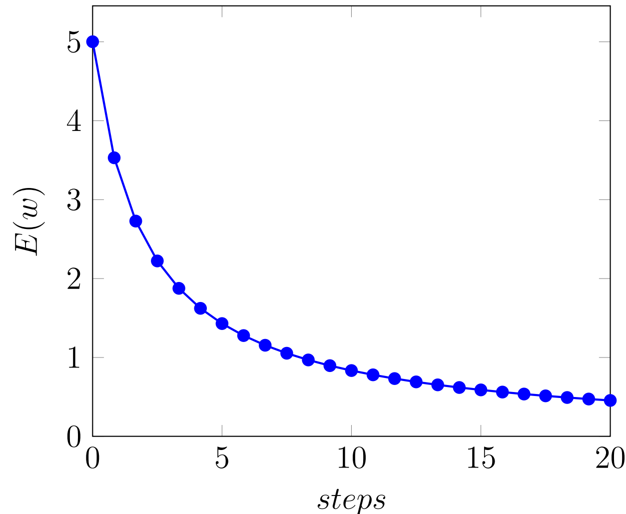


Figura 8: Relazione di $E(\mathbf{w})$ agli steps

L'algoritmo *gradient descent* così definito è particolarmente dispendioso dal punto di vista computazionale dovendo, per ogni passo, iterare sopra tutti gli elementi del *training set* il quale può raggiungere dimensioni consistenti. Per tale motivo sono state studiate diverse ottimizzazioni tra cui: Stochastic Gradient Descent, Adam e RMSProp. Di queste vengono illustrate solo quelle usate durante i test.

SGD) [10] *Stochastic Gradient Descent* rispetto al *gradient descent* per ogni aggiornamento dei pesi non valuta l'intero dataset per aggiornare i pesi. Per ogni esempio si aggiornano i pesi, ovvero da

$$\Delta w_i = \eta \sum_{j=0}^{k} (t^{(j)} - o^{(j)}) x_i^{(j)}$$

diventa

$$\Delta w_i = \eta (t^{(j)} - o^{(j)}) x_i^{(j)}$$

Il learning rate η nel caso dello SGD viene tipicamente preso inferiore rispetto al *gradient descent* in quanto vi è molta più varianza nell'aggiornamento.

Mini-Batch GD) Rispetto allo SGD e rispetto al *gradient descent* il quale considera l'intero dataset, anche chiamato col nome *batch gradient descent*, *mini-batch gradient descent* valuta sottoinsiemi, *batch* di esempi, per poi aggiornare i valori dei pesi.

SGD con momento) [10, 11] Se la funzione errore ha una forma di un burrone molto profondo e largo, lo SGD potrebbe oscillare tra le strette pareti e muo-

versi molto lentamente verso il centro. La convergenza all'ottimo può essere dunque particolarmente lenta.

Il momento è un metodo per spingersi più velocemente verso il centro e consiste nel tenere in considerazione i gradienti precedenti. Il momento v è:

$$\begin{aligned} v_{t,i} : &\quad \alpha v_{t-1,i} - \left.\frac{\partial E(\mathbf{w})}{\partial w_i}\right|_{(x^{(j)},y^{(j)})} \\ w_i : &\quad w_i + v_i \end{aligned}$$

con $v_{0,i} = 0 \quad i = 0, \ldots, m$. $\alpha \in (0, 1]$ rappresenta quanto influente è il momento passato rispetto al gradiente corrente.

SGD con momento di Nesterov)[11, 12] A differenza del momento calcolato in precedenza, il gradiente anzi che essere calcolato prima di considerare il momento, viene considerato dopo.

$$\begin{aligned} v_{t,i} : &\quad \alpha v_{t-1,i} - \left.\frac{\partial E(\mathbf{w} - \alpha \mathbf{v}_{t-1})}{\partial w_i}\right|_{(x^{(j)},y^{(j)})} \\ w_i : &\quad w_i + v_i \end{aligned}$$

RMSProp)[12] RMSprop divide il gradiente per la radice quadrata di una media mobile così definita:

$$S(w_i)_t = \beta S(w_i)_{t-1} + (1 - \beta) \left(\frac{\partial E}{\partial w_i}\right)^2$$

L'aggiornamento dei pesi diventa così:

$$w_i : w_i - \eta \frac{\frac{\partial E}{\partial w_i}}{\sqrt{S(w_i)}}$$

Ciò rende l'apprendimento più efficiente.

Adam)[12, 13] Adam è un algoritmo molto utilizzato. Esso unisce RMSProp e il momento.

Esso utilizza due medie mobili, ed alla t-esima iterazione:

$$V(w_i)_t = \beta_1 V(w_i)_{t-1} + (1 - \beta_1) \left(\frac{\partial E}{\partial w_i}\right)$$

$$S(w_i)_t = \beta_2 S(w_i)_{t-1} + (1 - \beta_2) \left(\frac{\partial E}{\partial w_i}\right)^2$$

$V(w_i)_t$ è la media che più assomiglia a quella vista per il momento, mentre $S(w_i)_t$ è la stessa media appena vista per l'RMSprop.

Di entrambe le medie se ne calcola la versione senza bias $\widehat{S}(w_i)_t$ ed $\widehat{S}(w_i)_t$, in quanto per le prime iterazioni t la media non rappresenta correttamente i dati. Ciò accade in quanto tenendo in considerazioni dati antecedenti, per le prime iterazioni la media con bias tende verso zero. Per t sufficientemente grandi invece $\hat{V}(w_i)_t \approx V(w_i)_t$ e $\hat{S}(w_i)_t \approx V(w_i)_t$

$$\widehat{V}(w_i)_t = \frac{V(w_i)_t}{1-\beta_1^t}$$

$$\widehat{S}(w_i)_t = \frac{S(w_i)_t}{1-\beta_2^t}$$

Queste versioni senza bias vengono utilizzate per aggiornare i pesi:

$$w_{i,t} : w_{i,t-1} - \eta \frac{\widehat{V}(w_i)_t}{\sqrt{\widehat{S}(w_i)_t} + \epsilon}$$

Dove η è il *learning rate*, ed ϵ è un termine molto piccolo il cui valore suggerito è 10^{-8} inserito affinché il denominatore non sia 0 ove $\sqrt{\widehat{S}(w_i)_t}$ o $\sqrt{\widehat{V}(w_i)_t}$ dovessero esserlo.

I valori consigliati per questo algoritmo sono:

$$\begin{aligned}
\eta &= 0.001 \\
\beta_1 &= 0.9 \\
\beta_2 &= 0.999 \\
\epsilon &= 10^{-8}
\end{aligned}$$

Le funzioni di attivazione f che sono state usate nel modello proposto sono due: la sigmoidea σ e la tangente iperbolica *tanh*:

$$\sigma(a) = \frac{1}{1+e^{-a}}$$

$$tanh(a) = \frac{1-e^{-2a}}{1+e^{-2a}}$$

La tangente iperbolica e la funzione sigmoidea sono legate dalla seguente relazione [8]:

$$tanh(a) = 2\sigma(2a) - 1$$

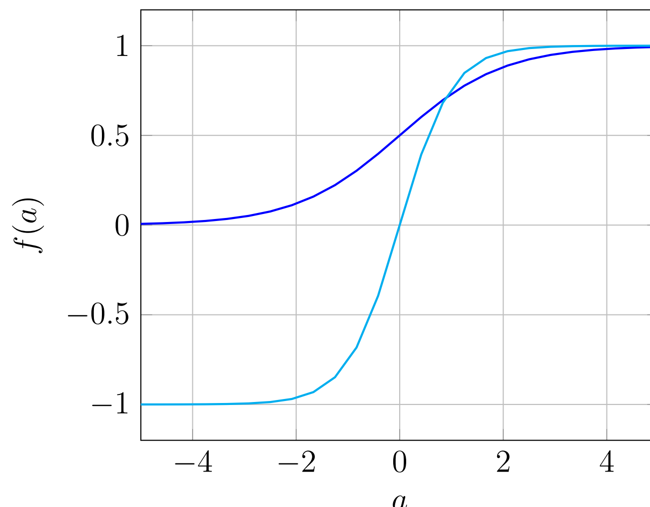


Figura 9: Sigmoide $\sigma(a)$ in scuro e tangente iperbolica $tanh(a)$ in chiaro. Si noti come $tanh : \mathbb{R} \rightarrow (-1, 1)$ rispetto alla $\sigma : \mathbb{R} \rightarrow (0, 1)$

Si possono applicare variazioni a $E(w)$. Per prima cosa definiamo una funzione di costo. Nel caso:

$$E(\mathbf{w}) = \frac{1}{k} \sum_{i=1}^{k} \frac{1}{2} (t^{(i)} - o^{(i)})^2$$

allora

$$cost(t, o) = \frac{1}{2}(t - o)^2$$

e quindi

$$E(\mathbf{w}) = \frac{1}{k} \sum_{i=1}^{k} cost(t^{(i)}, o^{(i)})$$

Questa funzione di costo MSE (*Mean Squared Error*) è utilizzata per esempio nella regressione lineare. Il modello introdotto ha il compito di classificare qualora una coppia faccia riferimento alla stessa entità reale, quindi più precisamente si tratta di classificazione binaria (vedi sezione 3) ed è quindi possibile applicare un modello di regressione logistica essendo t una variabile dicotomica e $o = p(t = 1 | x; \mathbf{w})$ e quindi la funzione di costo è la funzione logit:

$$cost(t, o) = -t \log(o) - (1 - t) \log(1 - o)$$

Che non è altro che una versione compatta di:

$$cost(t, o) = \begin{cases} -\log(o), & \text{if } t = 1 \\ -\log(1 - o), & \text{if } t = 0 \end{cases}$$

Inserendo questa funzione in $E(\mathbf{w}) = \frac{1}{k}\sum_{i=1}^{k} cost(t^{(i)}, o^{(i)})$ si ottiene:

$$E(\mathbf{w}) = -\frac{1}{k}\sum_{i=1}^{k}(t^{(i)}\log(o^{(i)}) + (1-t^{(i)})\log(1-o^{(i)}))$$

[14]

2.1.2 Strutture

Vengono di seguito illustrate le due strutture principali utilizzati nel modello successivamente illustrato ed impiegato nei test della sezione 4. Esse sono le reti neurali *feed-forward* e le reti neurali ricorrenti.

Come precedentemente indicato esistono anche altri tipi di struttura, le quali non vengono illustrate in quanto non utilizzate. Per esempio le reti neurali convoluzionali le quali sono molto utilizzate nel campo della visione artificiale in quanto adatte ad analizzare dati bi- o tri-dimensionali.[15].

2.1.2.1 Feed-forward Neural Network

Le reti neurali *feed-forward* sono tutte le reti neurali le cui connessioni non formano cicli. Di questa grande famiglia di interesse è il percettrone multistrato (*Multilayer Perceptron*, MLP).

Impilando i percettroni verticalmente si ottiene uno strato come in figura 10.

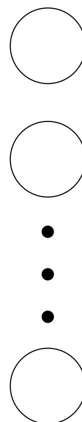


Figura 10: Strato di percettroni

I percettroni appartenenti allo stesso livello l non sono interconnessi. Collegando livelli orizzontalmente si ottiene un MLP, mostrato in figura 11.

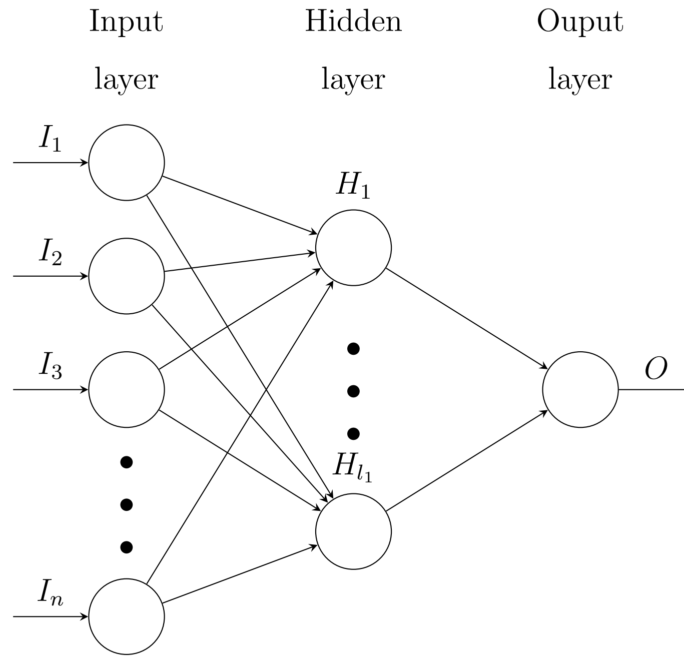


Figura 11: Percettrone multistrato

Ogni nodo è completamente connesso a tutti i nodi dello strato precedente e seguente, ovvero ogni nodo prende in ingresso tutte le uscite dei nodi dello strato precedente e l'uscita confluisce in ogni nodo dello strato seguente.

Chiamiamo L il numero totale dei layer, l_k il numero di unità al livello k. La funzione di attivazione della k-esima unità , dell'i-esimo strato l_i viene indicata con $a_k^{(i)}$. $W^{(i)} \in \mathbb{R}^{l_{i+1}\times(l_i+1)}$ è la matrice dei pesi dello strato i-esimo, ovvero mappa i pesi che uniscono lo strato $i-1$ con lo strato i, tale che $W_{j,k}^{(i)}$ rappresenti il peso riferito all'uscita del k-esimo nodo dello strato i-esimo in ingresso del j-esimo nodo nello strato $i+1$.

Il primo strato, chiamato strato di ingresso od *input layer*, non è realmente uno strato di percettroni bensì ogni nodo rappresenta una variabile del vettore ingresso $\mathbf{x}$. Nella figura di esempio l'ingresso è un vettore n dimensionale.

Tra lo strato di ingresso e lo strato di uscita ci possono essere diversi strati chiamati strati nascosti, *hidden layers*, ognuno con il proprio numero di unità l_k.

In figura non sono stati inseriti i termini di bias, essi possono essere rappresentati come nodi i quali, presenti in tutti gli strati eccetto lo strato di uscita, non prendono ingressi e la loro uscita, di valore unitario, è in ingresso a tutti i nodi dello strato successivo (eccetto il bias). Facendo riferimento alla figura, non sono stati rappresentati i termini $I_0 = 1$ ed $H_0 = 1$.

L'ultimo strato è chiamato banalmente strato di uscita, *output layer*. Prendendo in esempio la figura 11, partendo dall'uscita O:

$$O = y_1^{(2)} = a_1^{(2)}(\sum_{j=0}^{l_1} y_j^{(1)} w_{1,j}^{(1)})$$

e per quanto riguarda lo strato nascosto:

$$H_1 = y_1^{(1)} = a_1^{(1)}(\sum_{j=0}^{l_0=n} y_j^{(0)} w_{0,j}^{(0)})$$
$$H_2 = y_2^{(1)} = a_2^{(1)}(\sum_{j=0}^{l_0=n} y_j^{(0)} w_{0,j}^{(0)})$$
$$\vdots$$
$$H_{l_1} = y_{l_1}^{(1)} = a_{l_1}^{(1)}(\sum_{j=0}^{l_0=n} y_j^{(0)} w_{0,j}^{(0)})$$

quindi per il generico nodo k nell'i-esimo strato:

$$y_k^{(i)} = a_k^{(i)}(\sum_{j=0}^{l_{i-1}} y_j^{(i-1)} w_{k,j}^{(i-1)}) \quad i = 1, \dots, L-1 k = 1, \dots, l_i$$

Considerando che per $i = 0, \mathbf{y}^{(0)} = \mathbf{I}$ si può riscrivere l'equazione precedente in forma vettoriale, supponendo che la funzione di attivazione sia la medesima per ogni nodo nello stesso strato, supposizione che si concretizza nell'applicazione:

$$\mathbf{y}^{(i)} = a^{(i)}(\mathbf{y}^{(i-1)} \cdot W^{(i-1)}) \quad i = 1, \dots, L-1$$

Dato un insieme di dati $D = \{(x^{(0)}, t^{(0)}), (x^{(1)}, t^{(1)}), ..., (x^{(k)}, t^{(k)})\}$ e definita su di esso una funzione errore $E(\mathbf{w}) = \frac{1}{2}\sum_{i=0}^{k}(t^{(i)} - o^{(i)})^2$ e definito $z^{(i)} = \mathbf{y}^{(i-1)} \cdot W^{(i-1)} \quad i = 1, \dots, L-1$, il processo che dato un valore in ingresso ne calcola i risultati intermedi per poi giungere al risultato finale si chiama *forward propagation*.

L'algoritmo di *back propagation* consiste nel prendere il risultato della *forward propagation* ed aggiornare i pesi partendo dall'errore nello strato di uscita e propagare all'indietro gli errori allo scopo di modificare correttamente i pesi degli strati intermedi. Come indicato precedentemente, secondo l'algoritmo *gradient descent* e le sue varie ottimizzazioni, i pesi vengono modificati secondo $\frac{\partial E}{\partial w}$. Nel caso di un una struttura multistrato per ogni strato vi è una matrice di pesi $W^{(i)}$. Valutando la derivata di cui sopra si nota che dipende dalla somma z e dunque è possibile applicare la regola della catena:

$$\frac{\partial E}{\partial w_{j,k}^{(i)}} = \frac{\partial E}{\partial z_j^{(i)}} \frac{\partial z_j^{(i)}}{\partial w_{j,k}^{(i)}}$$

Introducendo:

$$\delta_j^{(i)} = \frac{\partial E}{\partial z_j^{(i)}}$$

$\delta_j^{(i)}$ sono spesso chiamati errori[8]. Per come $z^{(i)}$ è definito si può derivare che:

$$\frac{\partial z_j^{(i)}}{\partial w_{j,k}^{(i)}} = y_k^{(i-1)}$$

Quindi:

$$\frac{\partial E}{\partial w_{j,k}^{(i)}} = \delta_j^{(i)} y_k^{(i-1)}$$

Dato che per i nodi in uscita (ricordando che L è il numero di strati):

$$\delta_j^{(L-1)} = (y_j^{(L-1)} - t_j)$$

Per valutare i $\delta_j^{(i)}, 1 < i < L-1$ si fa ulteriormente uso della regola della catena.

$$\delta_j^{(i)} = \frac{\partial E}{\partial z_j^{(i)}} = \sum_{m=1}^{l_{(i+1)}} \frac{\partial E}{\partial z_m^{(i+1)}} \frac{\partial z_m^{(i+1)}}{\partial z_j^{(i)}} \quad 1 < i < L-1, \quad 1 < j < l_i$$

dalla quale sostituendo con i risultati precedentemente ottenuti e le definizioni di cui sopra risulta:

$$\delta_j^{(i)} = a'(z_j^{(i)}) \sum_{m=0}^{l_{i+1}} w_{m,j}^{(i)} \delta_m^{(i)}$$

la quale rende l'errore di del j-esimo nodo dell'i-esimo strato. Questo viene applicato all'algoritmo di *gradient descent* od a qualunque delle versioni ottimizzate precedentemente illustrate. [5, 8, 14]

2.1.2.2 Recurrent Neural Network

Le reti neurali ricorrenti sono reti nelle quali sono presenti cicli. Esse sono molto efficaci nel rappresentare informazioni che si presentano in sequenza. Nel nostro caso essendo i dati sotto forma di testo potrebbe essere interessante valutarlo sequenzialmente in quanto parte dell'informazione è contenuta nella specifica sequenza.

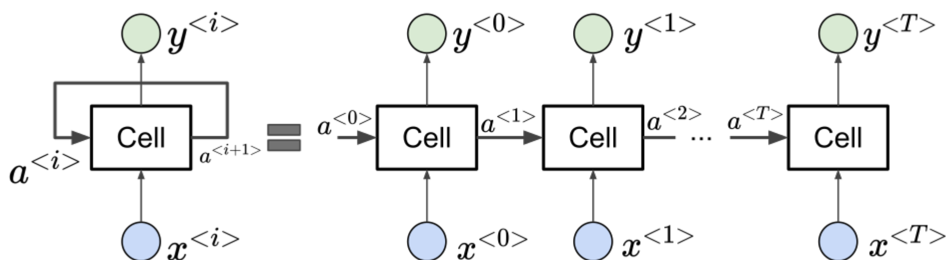


Figura 12: Rete neurale ricorrente

Come mostrato in figura 12 la più semplice struttura di una rete neurale ricorrente consiste di una cella la quale oltre ad avere un ingresso ed un uscita ha una terza uscita che è a sua volta un ingresso stesso. Tale struttura può essere districata in una struttura lineare ove $< i >$ rappresenta un istante temporale. La cella più semplice consiste in due percettroni, essendo due le uscite, le cui equazioni sono:

$$\begin{aligned}\mathbf{a}^{<i+1>} &= g_1(W_{aa}\mathbf{a}^{<i>} + W_{ax}\mathbf{x}^{<i>}) \\ \mathbf{y}^{<i>} &= g_2(W_{ya}\mathbf{a}^{<i>})\end{aligned}$$

Considerando $\mathbf{a}^{<0>} = \mathbf{0}$ Solitamente le funzioni di attivazioni g_1 e g_2 sono rispettivamente tanh e σ [16].

Passando attraverso l'operazione di srotolamento (*unravel*), la struttura derivante ricorda quella del percettrone multistrato illustrato precedentemente. Viene anche qui impiegato l'algoritmo di *backpropagation*, che essendo applicato ad istanti temporali in successione viene chiamato *backpropagation through time*, BPTT. Anziché esserci i vari strati i vi saranno vari istanti di tempo t. Dunque l'errore valutato all'istante di tempo t viene propagato all'indietro in riferimento agli istanti di tempo precedenti in maniera analoga a quella indicata precedentemente. Essendo i diversi pesi i medesimi per ogni istante temporale, vengono calcolati i ΔW per ogni istante e poi sommati per modificare i pesi[17].

Un problema che può presentarsi con questo tipo di implementazione (una cella con due percettroni) è che per sequenze particolarmente lunghe nel tempo i gradienti tendono ad essere infintesimi, a svanire, e per tale ragione è definito *vanishing gradient*. Questo significa che una RNN così costruita fatica a codificare dipendenze lontane temporalmente.

Per contrastare il *vanishing gradient* vengono usate delle celle con una struttura particolare le quali contengono metodi per costruire una memoria la quale viene anch'essa propagata.

Una di queste unità particolari è la *Long Short-Term Memory unit* introdotta nel 1997 da Sepp Hochreiter ed Jürgen Schmidhuber[18] la quale oltre ad avere $x^{<t>}, a^{<t>}$ ed $y^{<t>}$ presenta $c^{<t>}$. Presenta tre cancelli, *gate*, i quali regolano modifiche alla memoria od all'uscita.

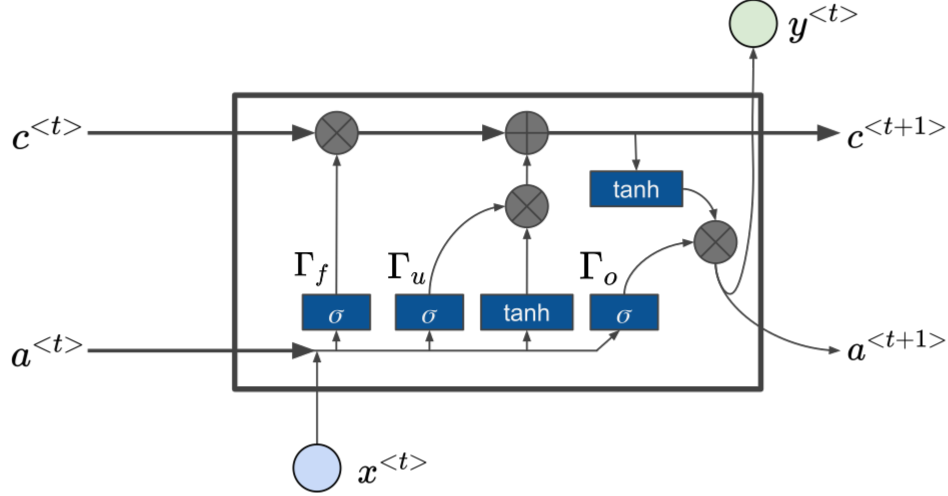


Figura 13: Unità LSTM

Infatti $c^{<t+1>}$ che rappresenta la memoria che viene propagata in avanti è calcolato come:

$$c^{<t+1>} = \Gamma_f \cdot c^{<t>} + \Gamma_u \cdot \tilde{c}^{<t>}$$

dove Γ_f e Γ_u sono i *gate* che indicano rispettivamente quanto dimenticare e quanto aggiornare con nuovi valori di memoria $\tilde{c}^{<t>}$. Questi *gate* sono computati come segue:

$$\Gamma_f = \sigma(W_f[\mathbf{a}^{<t>}, \mathbf{x}^{<t>}])$$
$$\Gamma_u = \sigma(W_u[\mathbf{a}^{<t>}, \mathbf{x}^{<t>}])$$
$$\Gamma_o = \sigma(W_o[\mathbf{a}^{<t>}, \mathbf{x}^{<t>}])$$

$\tilde{c}^{<t>}$ che rappresenta quello che si vorrebbe memorizzare all'istante t è così calcolato:

$$\tilde{c}^{<t>} = \tanh(W_c[a^{<t>}, x^{<t>}])$$

ed è usato per ricavare $c^{<t+1>}$ come sopra. Il *gate* Γ_o è utilizzato per ottenere lo stato futuro e l'uscita. Constatando che $y^{<t+1>} = a^{<t+1>}$ e considerando che con $[a^{<t>}, x^{<t>}]$ si intende il vettore ottenuto concatenando $a^{<t>}$ e $x^{<t>}$, allora:

$$a^{<t+1>} = y^{<t+1>} = \Gamma_o \cdot \tanh(c^{<t>})$$

Con questi diversi *gates*, se questi sono correttamente impostati, le celle LSTM possono codificare dipendenze molto separate temporalmente facendo passare informazioni tramite la memoria c [16].

Un ulteriore problema che si ha con il modello di RNN illustrato è che catturano informazioni in un'unica direzione essendo anche chiamate *unidirectional RNN* od anche *forward directional RNN*. Ovvero ad un punto t di una sequenza incapsulano esclusivamente i dati precedenti e non hanno a disposizione informazioni seguenti nella sequenza. A tale scopo vi sono le *bidirectional RNN*, BRNN. Esse ottengono questo risultato facendo scorrere la sequenza in entrambe le direzioni. Da ciò si ottengono due risultati, uno da una direzione ed uno dall'altra le quali possono poi essere combinate in diversi modi, per esempio: concatenandole, sommandole o facendone la media.

2.2 Support Vector Machine

Le *Support Vector Machines*, o SVM, sono un insieme di algoritmi di *supervised learning* di classificazione, i quali intuitivamente mirano a massimizzare il margine tra il piano di separazione ed i punti più vicini. Per questo vengono definite anche come *large margin classifiers*.

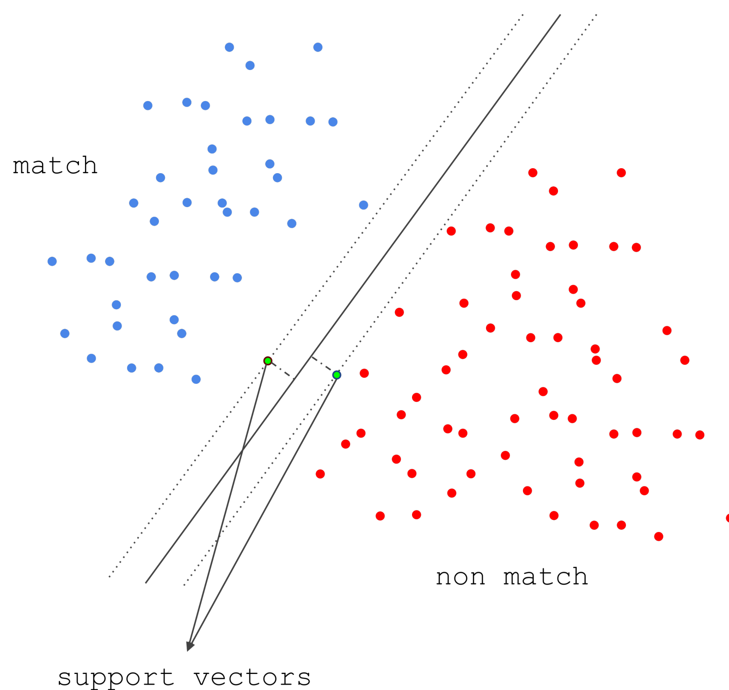


Figura 14: Esempio di SVM

Senza entrare nella spiegazione matematica di ciò che avviene e quindi limitandosi ad una comprensione intuitiva, supponendo che l'insieme dei dati, del *training set*, sia linearmente separabile allora l'SVM trova quell'iperpiano il quale massimizza il margine rispetto ai punti più vicini, i quali si chiamano *Support Vectors*. I *support vectors* sono i quelli per cui, dovessero variare minimamente, varierebbe anche l'iperpiano di separazione.

L'SVM porta con se diversi vantaggi. Innanzi tutto non dovendo operare con l'interezza dei dati, ma solo coi *support vectors*, può essere scalato a problemi il cui *training set* è molto vasto senza subire un notevole impatto prestazionale in termini di risorse computazionali e temporali. Inoltre l'SVM è particolarmente efficiente in spazi con alte dimensioni [19]. Nel caso i dati non fossero linearmente separabili, si possono impiegare dei *kernel*. Essi mappano i punti in un nuovo spazio nel quale dovrebbero essere linearmente separabili. Esistono diversi *kernel*, di particolare interesse è quello polinomiale, che tiene in considerazioni combinazioni dei dati nella forma $(\mathbf{x}^T x_i + r)^d$ [14].

Uno dei principali difetti dell'SVM, il quale però non influenza l'algoritmo illustrato di seguito è che l'uscita è, per quanto riguarda la classificazione binaria esclusivamente 1 o 0, mentre i metodi precedentemente riportati con le reti neurali permettono di ottenere le probabilità rispetto alla decisione o valutazione effettuata [19].

3 Algoritmo

Preso in considerazione il problema da affrontare, illustrato nella sezione 1, e le diverse tecniche di machine learning, mostrate nella sezione 2, viene spiegato ora come queste ultime vengano combinate ed applicate al suddetto problema.

3.1 Deeper Model

Il modello preso in considerazione è quello descritto in "Distributed Representations of Tuples for Entity Resolution" di Muhammad Ebraheem et al [1].

L'obbiettivo di questo modello è presa una coppia di tuple codificare la similitudine tra esse e successivamente classificare correttamente quelle che fanno riferimento alla stessa entità reale. Il primo passo è rappresentare numericamente una tupla, per far ciò, bisogna prima di tutto avere una rappresentazione numerica di una parola, detta *word embedding*, dalla quale si può ottenere una rappresentazione numerica di un attributo o di una tupla.

3.1.1 Average

Supponendo di avere una rappresentazione numerica di una parola, ovvero per una parola supponiamo di avere un vettore $x \in \mathbb{R}^k$, dove k è la dimensione dell'*embedding*, allora il metodo più intuitivo per ottenere una rappresentazione numerica di un valore di attributo, il quale è una combinazione di parole, è calcolando la media delle rappresentazioni delle parole in esso contenute.

Algorithm 1 Average

for ogni attributo A_k in t **do**
  for ogni parola p in A_k **do**
    ottenere la rappresentazione numerica della parola p: $n_p \in \mathbb{R}^k$
  ottenere la media delle rappresentazioni numeriche $\bar{n}_k$
Concatenare le varie $\bar{n}_k$ in modo da ottenere una singola rappresentazione di una tupla $V_t \in \mathbb{R}^{m \times k}$, con m il numero degli attributi A_k

Data una coppia di tuple, l'algoritmo appena mostrato viene applicato ad entrambe. Per ogni riga in V_i e V_j ne viene calcolata la cosine similarity. Essa è una misura di similarità spesso applicata per rappresentare la similitudine tra due vettori.

$$similarity(\mathbf{a}, \mathbf{b}) = \cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|}$$

Dunque sia $\mathbf{x}_{(i,j)}$ il vettore contenente i vari valori di similitudine tra i vari attributi:

$$\mathbf{x}_{(i,j)} = \begin{bmatrix} \text{similarity}(V_{i,1}, V_{j,1}) \\ \text{similarity}(V_{i,2}, V_{j,2}) \\ \vdots \\ \text{similarity}(V_{i,m}, V_{j,m}) \end{bmatrix}$$

Notare che $V_{t,h} \in \mathbb{R}^k \quad 0 < h \leq m$.

Questo vettore viene usato come input di una SVM o di una rete neurale con uno strato nascosto ed una uscita, essendo un problema di classificazione binaria, come mostrato in figura 15.

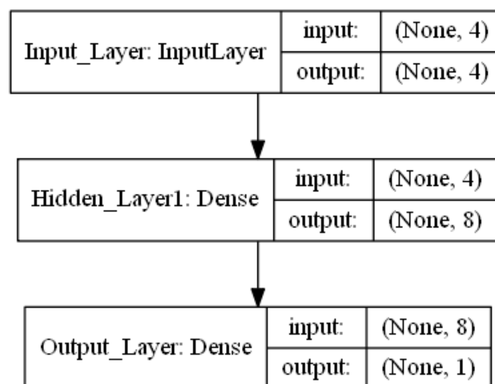


Figura 15: Neural Network usata con la media. L'ingresso ha forma *(None,4)* dove *None* sostituisce la dimensione del *training set* e 4 è il numero di attributi in esempio. Lo strato nascosto ha in esempio 16 unità.

3.1.2 LSTM

Il modello appena indicato, nonostante sia più semplice da istruire, ignora la sequenza di parole. A tal proposito viene suggerito di utilizzare una rete neurale ricorrente con celle LSTM le quali hanno lo scopo di codificare le informazioni contenute nella sequenza. Ci possono essere diverse combinazioni in cui si possono applicare le LSTM:

1. una rete per ogni attributo di ogni tupla (due volte il numero di attributi);
2. una rete per ogni coppia di attributi delle tue tuple (il numero di attributi);
3. una rete per ogni tupla (due), come mostrato in figura 16;
4. una rete, condivisa tra le due tuple, come mostrato in figura 17.

Il modello DeepER suggerisce di usare LSTM condivise, ovvero i cui pesi sono condivisi, tra gli attributi, il che esclude l'1. Procedendo nell'analisi del documento si evince che anche il tipo di modello 2 non è quello utilizzato, ma rimane non specificato quale tra il 3 ed il 4 siano il modello suggerito. Dunque nella fase di test (vedi 4) verranno utilizzati entrambi[1].

I dati in uscita alle LSTM (in figura sono 300 dimensionali), vengono composti tramite il prodotto di hadamard, ossia elemento per elemento. Questo vettore così composto è poi utilizzato come input ad una rete neurale simile a quella nel modello precedente.

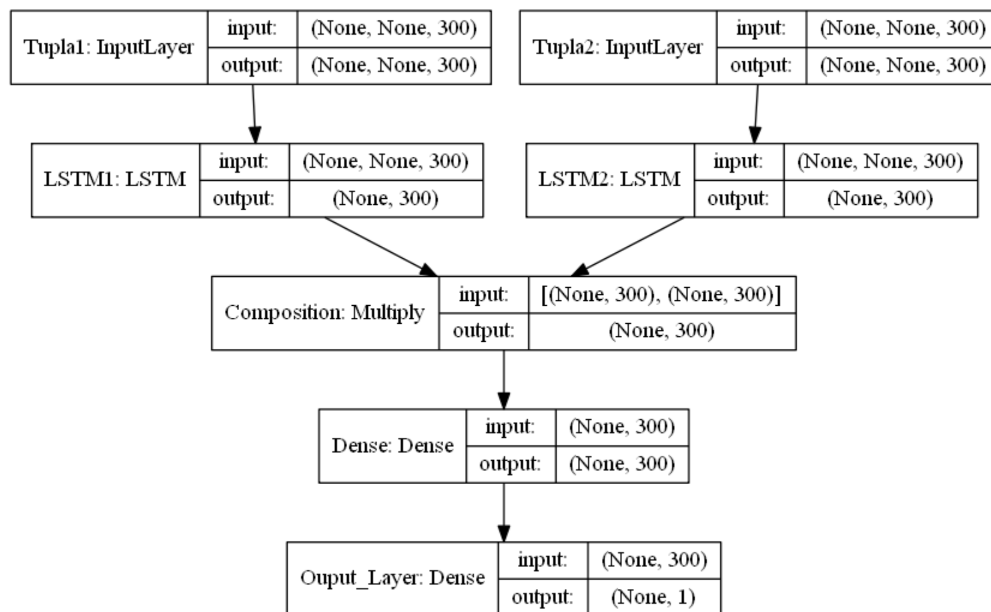


Figura 16: Doppia LSTM

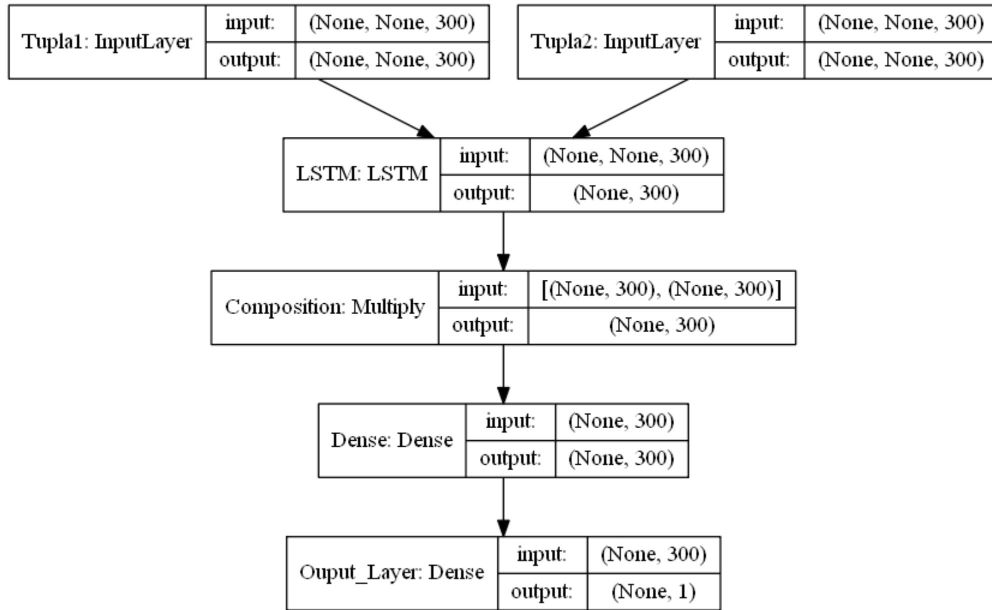


Figura 17: Singola LSTM

Nelle figure i due *Input Layer*, che sono indicati con Tupla1 e Tupla2 rispettivamente, sono le rappresentazioni numeriche di due tuple. La dimensione *(None, None, 300)* sta a significare *(numero_di_esempi, numero_di_parole, dimensione_dell_embedding).* Il numero di esempi è variabile così come il numero di parole il quale però dev'essere uguale per ogni gruppo di parole che viene fornito, è quindi presente del *padding* di zeri il quale può essere anteposto o posposto. Ogni LSTM restituisce un unico vettore di dimensione d, in figura 300. Essendo due gli ingressi ne segue che saranno presenti due vettori d dimensionali i quali vengono poi composti tramite prodotto di hadamard risultando sempre in un vettore d dimensionale. Questo vettore è ingresso di uno strato di h unità, in figura 300, il quale non deve essere necessariamente uguale alla dimensione del risultato dell'LSTM. All'uscita vi è un unico percettrone.

3.1.3 Bi-LSTM

Le LSTM appena illustrate e come indicato nella relativa sezione (2.1.2.2) codificano l'informazione da sinistra verso destra, in maniera unidirezionale. Per ciò vengono impiegate anche delle Bi-LSTM, ovvero LSTM bidirezionali. Queste dato un attributo danno origine a due vettori $\overrightarrow{h_k}$ e $\overleftarrow{h_k}$ i quali rappresentano rispettivamente l'informazione tramite un passaggio da sinistra verso destra e viceversa. Questi sono poi combinati tramite concatenazione per creare un unico vettore $[\overrightarrow{h_k}, \overleftarrow{h_k}]$ rappresentante la tupla.

Per il resto nulla varia rispetto al modello con le LSTM.

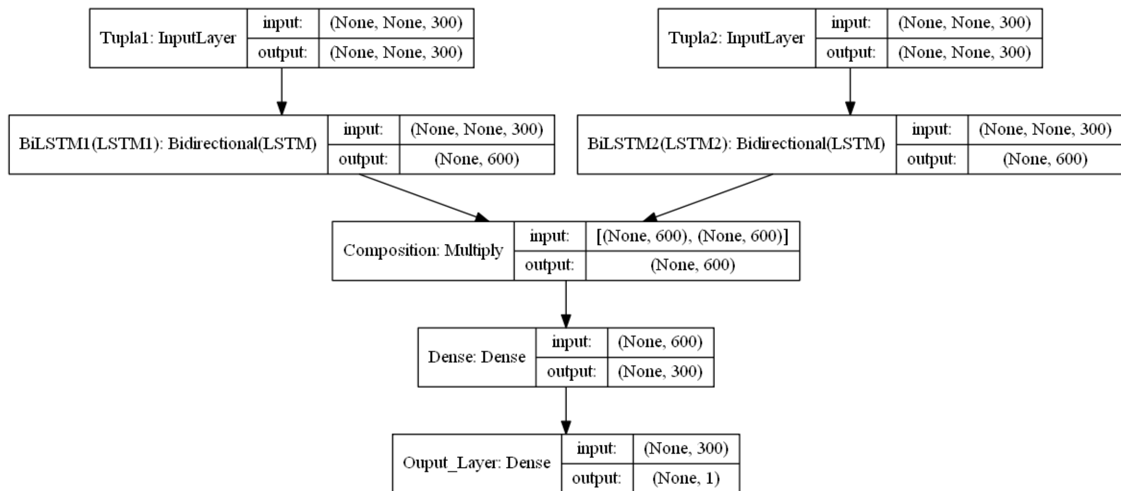


Figura 18: Doppia Bi-LSTM

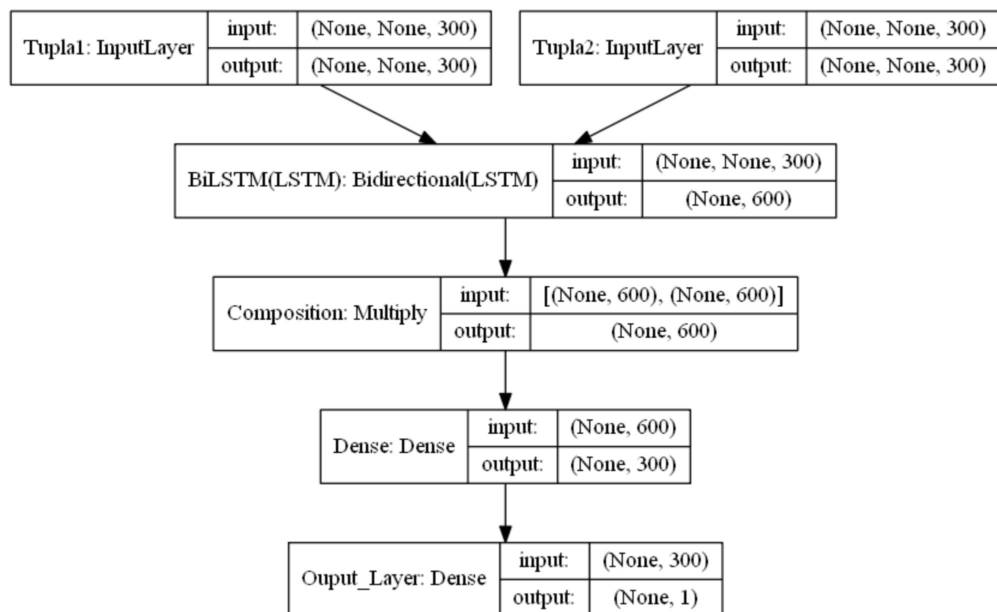


Figura 19: Singola Bi-LSTM

Come si può notare nelle figure, l'uscita delle Bi-LSTM e 600 dimensionale, anche se la dimensione dell'LSTM sottostante non è espressamente indicata, è identica a quella della LSTM nell'esempio precedente, 300 dimensionale. Notare che la figura può essere fuorviante, come appena indicato la dimensione dell'uscita della Bi-LSTM non dipende dalla dimensione dell'ingresso.

3.2 Word Representation

Rappresentare una parola numericamente non è un problema banale. Una parola è inserita in un determinato contesto e può assumere un diverso significato. Se non volessimo rappresentare il contesto ma solo identificare la parola in quanto tale, si potrebbe pensare di creare un dizionario di tutte le parole nel *dataset*, e poi codificarle con i vettori onehot. Questi vettori possiedono un uno in una posizione e zero in tutte le altre. In questo caso il vettore rappresentante la parola avrebbe come dimensione il numero di singole parole all'interno del *dataset*.

$$parola = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \in \mathbb{R}^{dimensione\ del\ dizionario}$$

Le due critiche più rilevanti che vengono avanzate a questo modo di rappresentare una parola è che:

- non incapsulano nessun tipo di informazione che riguarda il contesto,
- per *dataset* sufficientemente grandi il cui dizionario è sufficientemente vario la dimensione di questi vettori rischia di diventare eccessivamente grande, il che la rende una soluzione non scalabile.

Per insiemi di parole piccole e di *dataset* piccoli questi non sono particolarmente rilevanti ma per *dataset* grandi questi diventano particolarmente rilevanti, il che la rende una soluzione non scalabile. Per tale ragione si cerca una soluzione scalabile.

3.2.1 Pre-made embeddings

Una soluzione interessante sono *embeddings* precompilati, ovvero sono dizionari che data una parola ne rendono (se è presente nel dizionario) una rappresentazione numerica la cui dimensione è fissata. Il grosso vantaggio di queste rappresentazioni è che non richiedono di essere sviluppate sul *dataset* e quindi non richiedono tempo che rappresentazioni sviluppate *ad hoc* richiederebbero. Questi *embeddings* già preparati hanno

però un insieme di parole limitato il che significa che è possibile che un certo numero di parole non sia incluso nel dizionario. Questo numero dipende da quanto specifico è il vocabolario del *dataset* preso in considerazione.

Esistono dizionari di diversa dimensione, sia in numero di parole comprese che come dimensione della rappresentazione numerica.

3.2.1.1 GloVe

Uno dei modelli più utilizzati è il "Global Vectors for Word Representation" di Jeffrey Pennington, Richard Socher e Christopher D. Manning [20].

Questo modello cerca di trovare una rappresentazione d dimensionale fissata ottimizzata rispetto alle probabilità di correlazione incrociata, supponendo che in tali probabilità sia contenuta dell'informazione. Questo viene studiato su corpus molto grandi. Nei test illustrati di seguito è stato scelto il vocabolario modellato su Wikipedia 2014 e Gigaword 5, con un vocabolario di 400 000 parole codificate con vettori 300 dimensionali.

Questo tipo di modello restituisce parole del tipo:

$$parola = \begin{bmatrix} 1.345 \\ -0.365 \\ \vdots \\ 3.24 \\ 0.1 \end{bmatrix} \in \mathbb{R}^{300}$$

4 Test

In questa sezione verranno illustrati i vari *dataset* ed i vari risultati ottenuti su di essi con le diverse tecniche appena illustrate.

Innanzi tutto consideriamo che un certo *dataset* contenga h tuple allora si potrebbero considerare tutte le coppie possibili $C'_{h,2} = \binom{h+2-1}{2} = \binom{h+1}{2}$. Considerando però che il numero di *match*, ovvero delle copie che combaciano, sia molto inferiore a $\binom{h+1}{2}$. Dunque si procede a costruire un sottoinsieme di $\binom{h+1}{2}$ nel seguente modo: tra tutte le coppie che fanno *match* ne viene calcolato il minimo della similarità: $\min\{simlarity(i,j)|(i,j) \in \binom{h+1}{2}, s(i,j) = true, i \neq j\}$, questo minimo è utilizzato come soglia nella scelta delle coppie che non sono *match*, nel caso non esistesse una coppia che non corrisponda a tale criterio viene scelta casualmente. Il *dataset* è costruito unendo tutte le coppie che fanno *match*, includendo anche le coppie costituite da una tupla con se stessa, e due coppie *mismatch* per ognuna delle prime. Questo è poi suddiviso un tre parti:

- *training set* 60%: utilizzato per apprendere come separare correttamente le coppie;
- *validation set* 20%: utilizzato per valutare i diversi iperparametri (learning rate, numero di unità, funzione di attivazione, algoritmo, ecc.);
- *test set* 20%: utilizzato infine sul modello migliore.

4.1 Fodor's Zagat

Il Fodor's e il Zagat [2] sono due guide di ristoranti di cui alcune tuple sono mostrate in tabella 3. Questo insieme di ristoranti preso in considerazione è piuttosto ristretto, contiene soltanto 864 tuple con 112 match (escludendo i *match* causati dalle tuple con se stesse, che sono ovviamente 864) considerando che $h = 864$, $\binom{h+1}{2} = 372816$ il rapporto:

$$\frac{matches}{\binom{h+1}{2}} = \frac{112 + 864}{372816} = 0.262\%$$

Questo per mostrare come considerare tutte le coppie creerebbe una disparità troppo rilevante tra *match* e *non-match*. Col metodo sopra indicato si ottengono un totale di 2928 coppie.

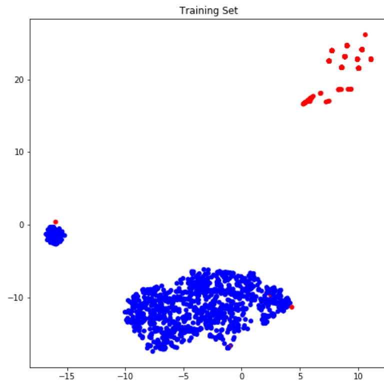
Nome	Indirizzo	Città	Tipo
arnie morton's of chicago	435 s. la cienega blv.	los angeles	american
lespinasse	2 e. 55th st.	new york	american
lespinasse (new york city)	2 e. 55th st.	new york city	asian
gianni's	5 fulton st.	new york	seafood

Tabella 3: Alcune tuple dal Fodor's Zagat

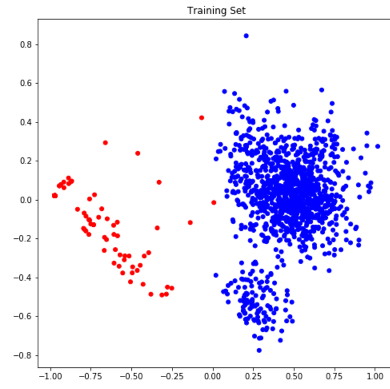
Usando tecniche di riduzione delle dimensioni si può rappresentare in due dimensioni quello che avviene in più dimensioni. Usando la tecnica della media, si ottiene un vettore 4 dimensionale. Tramite una delle tecniche di riduzione della dimensionalità, la *t-Distributed Stochastic Neighbor Embedding*, o t-SNE [21], si vuole dare una idea dello spazio delle coppie allo scopo di comprendere il livello di difficoltà del dataset situata nella separazione delle coppie *match* e *mismatch*. La t-SNE è una trasformazione non lineare che vuole mantenere la maggior parte della struttura presente nello spazio a dimensionalità maggiore nella rappresentazione a dimensionalità minore. L'altra tecnica mostrata, la *Principal Component Aalysis*, o PCA, è una trasformazione lineare la quale cerca le direzioni le cui proiezioni dei dati hanno massima varianza [22]. Nel caso di dimensione finale uguale a due come nel nostro caso si considerano le prime due direzioni a varianza massima. Questa rappresentazione permette di apprezzare la separabilità lineare dei dataset, trattandosi di una trasformazione lineare. In blu sono rappresentate le coppie *non-match* ed in rosso le coppie *match*.

Come si nota in figura 20 i dati sono abbastanza separati, infatti i modelli più semplici riescono a distinguerli adeguatamente. I grossi raggruppamenti circolari individuabili nei t-SNE (sia di questo che degli altri dataset) contengono le coppie formate da una tupla con se stessa.

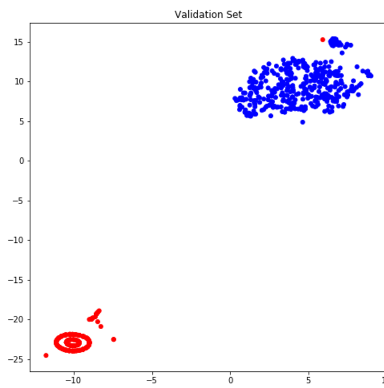
I risultati seguenti sono i migliori che sono stati ottenuti variando i vari parametri per cercarne di migliorare il risultato sul *validation set*.



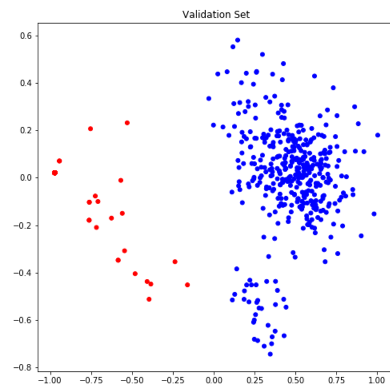
(a) Training Set t-SNE



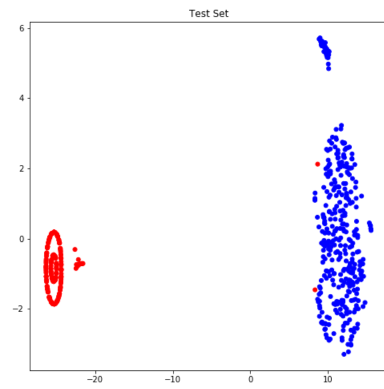
(b) Training Set PCA



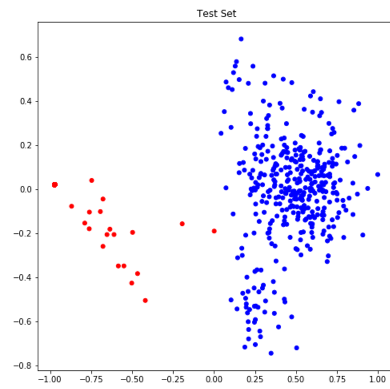
(c) Validation Set t-SNE



(d) Validation Set PCA



(e) Test Set t-SNE



(f) Test Set PCA

Figura 20: t-SNE e PCA applicata ad ognuna delle tre partizione del dataset

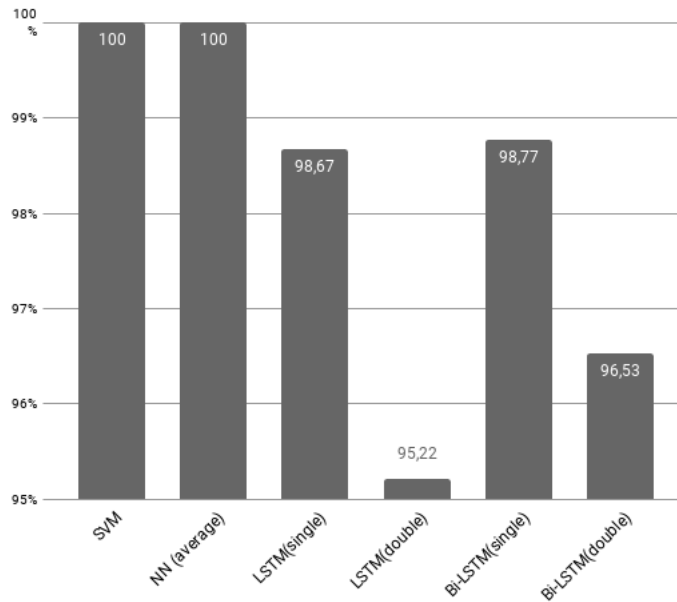


Figura 21: Fodor's - Zagat F_1-score

Come si evince dalla figura 21 sia la SVM che la rete neurale semplice, nei quali le coppie sono rappresentate tramite il metodo della media, riescono a modellare eccezionalmente i dati del *validation set*, dati che non sono stati visti durante la fase di training. Tutti i modelli che includono algoritmi ricorrenti invece hanno un risultato peggiore, dovuto probabilmente sia dalla dimensione ridotta dell'insieme di dati che dalla sua semplicità. É dunque probabile che vi sia *overfitting*, ovvero questi modelli descrivono "troppo" bene i dati del training set e non riescono a generalizzare sufficientemente bene su dati ignoti. Oltre che ad essere meno performanti, richiedono anche un tempo di training molto maggiore rispetto alle prime come illustrato in figura 22.

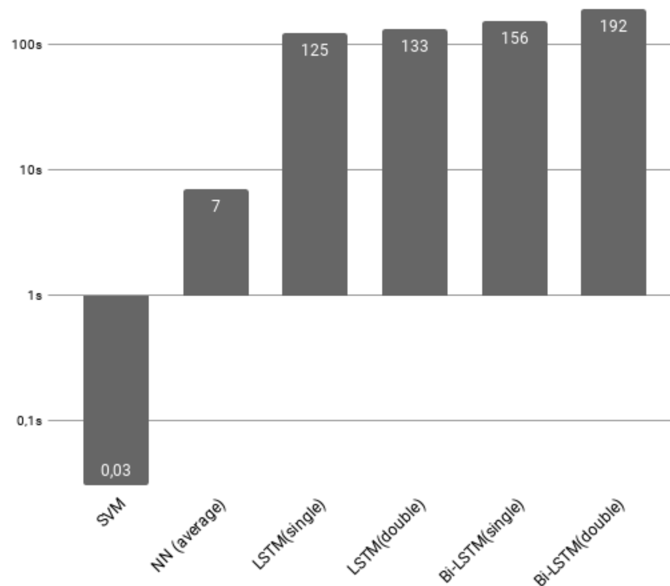


Figura 22: Fodor's Zagat tempo di addestramento in secondi con una scala logaritmica

Rilevante è invece notare come la versione con una singola cella (sia LSTM che Bi-LSTM), sia migliore di quella a cella doppia.

Valutando il test set sia sulla SVM migliore che sulla NN migliore si ottengono i seguenti risultati:

	Precision	Recall	F_1
NN(average)	100%	100%	100%
SVM	100%	99.51%	99.76%

Possiamo quindi concluderne che per piccoli insiemi di dati, poco complessi, che presentano eventuali elementi sporchi la rete neurale che usa l'algoritmo della media riesce a dare un risultato ottimo.

4.2 DBLP ACM

Il DBLP ACM[3] contiene riferimenti bibliografici, rispettivamente associati a *DBLP computer science bibliography* e *Association for Computing Machinery*, di cui se ne può apprezzare un estratto in tabella 4. Questo dataset aumenta la propria complessità rispetto al Fodor's Zagat:

- includendo sequenze di parole più lunghe: titoli, nomi di autori;

- utilizzando un vocabolario più specifico, una percentuale maggiore di parole finirà per essere rappresentata con la chiave *"unk"*[2];

- sono presenti una maggior quantità di dati, ciò rende più semplice far apprendere all'algoritmo il confine tra *match* e *non match*, ma richiederà più tempo di addestramento.

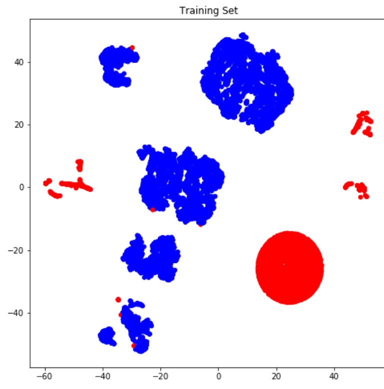
Inoltre vi sono elementi particolarmente sporchi come l'ultimo nell'esempio di tabella 4.

In questo dataset vi sono 4910 tuple e delle $\binom{4910+1}{2} = 12056505$ coppie possibili vi sono 2224 *match* La percentuale dei *match* su tutte le coppie possibili è:

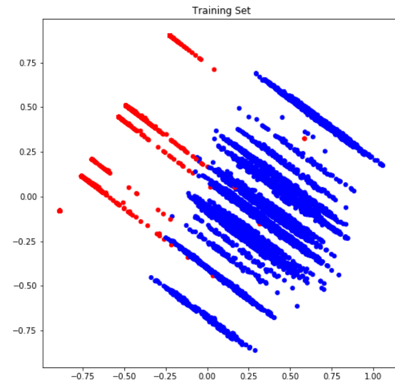
$$\frac{matches}{\binom{h+1}{2}} = \frac{2224 + 4910}{12056505} = 0.0592\%$$

Con la tecnica utilizzata si ottengono un numero di coppie complessive pari a 21402.

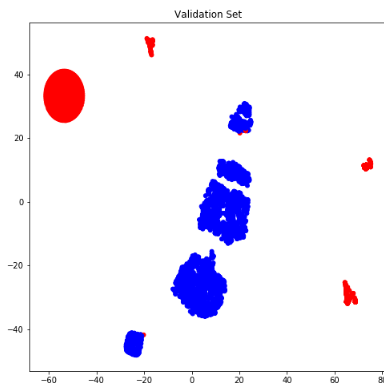
[2]La stringa *"unk"* è presente nel dizionario GloVe e rappresenta un vettore che viene utilizzato per rappresentare le parole che non sono contenute all'interno del dizionario stesso



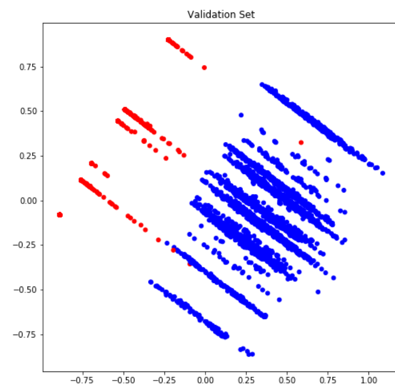
(a) Training Set t-SNE



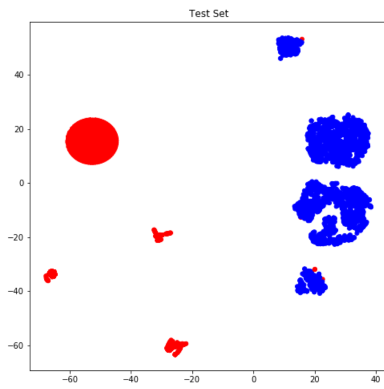
(b) Training Set PCA



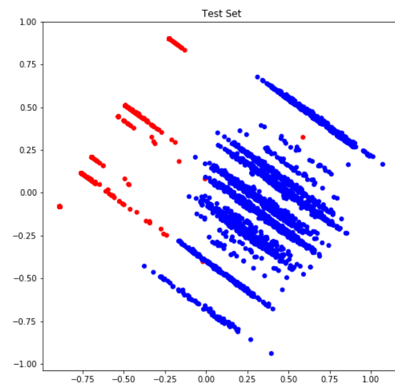
(c) Validation Set t-SNE



(d) Validation Set PCA



(e) Test Set t-SNE



(f) Test Set PCA

Figura 23: t-SNE e PCA applicata ad ognuna delle tre partizione del dataset

Titolo	The WASA2 object-oriented workflow management system
Autore	Gottfried Vossen, Mathias Weske
Origine	International Conference on Management of Data
Anno	1999
Titolo	DOMINO: databases fOr MovINg Objects tracking
Autore	Ouri Wolfson, Prasad Sistla, Bo Xu, Jutai Zhou, Sam Chamberlain
Origine	International Conference on Management of Data
Anno	1999
Titolo	DOMINO: Databases fOr MovINgObjects tracking
Autore	Jutai Zhou, Ouri Wolfson, Sam Chamberlain, A.Prasad Sistla, Bo Xu
Origine	SIGMOD Conference
Anno	1999
Titolo	Title
Autore	?
Origine	VLDB J.
Anno	1995

Tabella 4: Esempio di tuple del DBLPACM

Usando le stesse tecniche impiegate precedentemente si rappresentano lo spazio dei dati in figura 23. Si può ancora notare come i dati siano sufficientemente separati a parte qualche punto rosso che si può notare essere contenuto nella zona a maggioranza blu. Come illustrato a breve vi sono diversi errori ricorrenti (probabilmente questi punti) i quali vengono erroneamente classificati a causa di un vocabolario ristretto.

I risultati migliori ottenuti modificando i parametri valutando progressivamente i risultati sul validation set sono illustrati in figura 24

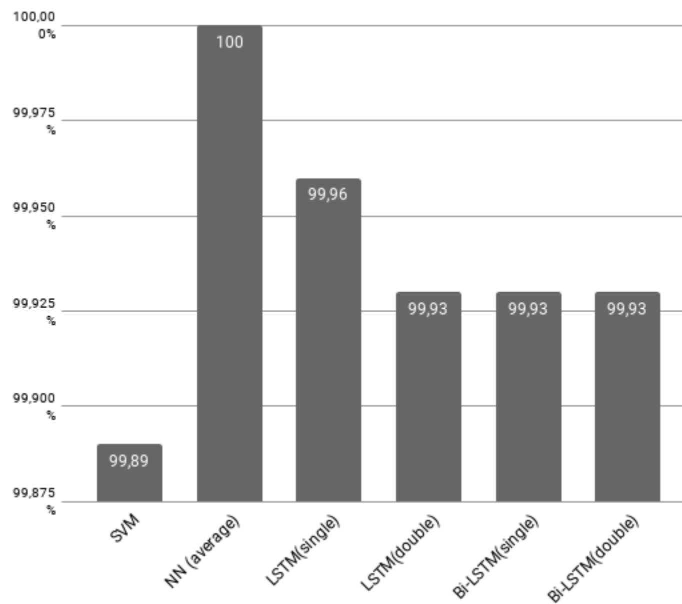


Figura 24: DBLP ACM F_1-score

Come si nota la SVM, che nel Fodor's Zagat primeggiava, diventa la meno performante. Ciò è dovuto all'aumento della complessità dei dati i quali la SVM fatica a separare correttamente.

La semplicità della rete neurale che usa la tecnica della media illustrata nella sezione 3.1.1, le permette di generalizzare perfettamente sui dati ignoti del validation set ed ottenere un risultato ottimo.

Le differenze notate per il Fodor's Zagat sugli algoritmi ricorrenti vengono limate, a causa dell'ampliamento dei dati disponibili. Primeggia però, anche se con differenza di qualche centesimo di punto percentuale l'algoritmo che include una singola rete LSTM.

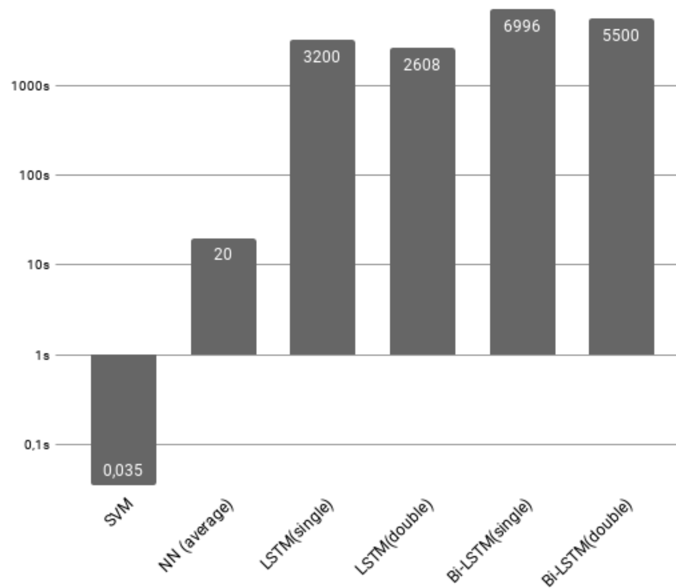


Figura 25: DBLP ACM tempo di training in secondi su scala logaritmica

Nonostante questi risultati, i tempi che hanno impiegato i vari algoritmi per il training sono molto rilevanti. Come illustrati in figura 25 in scala logaritmica, e come notato per il Fodor's Zagat i tempi per i modelli contenenti reti ricorrenti sono notevolmente maggiori, rendendoli assai meno appetibili anche a fronte di possibili risultati migliori, che comunque non si sono ancora presentati.

Sottoponendo alla rete neurale il test set, che non è mai stato visionato si ottiene il seguente risultato:

	Precision	Recall	F_1
NN(average)	100%	99.86%	99.93%

Di interesse sono i due errori più comuni che si sono presentati tra tutti i modelli. Sono di particolare interesse in quanto contengono diverse parole specifiche le quali non sono presenti nel vocabolario e che vanno quindi a sporcare la rappresentazione numerica complessiva, rendendone difficile la corretta identificazione. Un vocabolario differente potrebbe risolvere il problema migliorando la rappresentazione riducendo il rumore introdotto dalle parole fuori dal vocabolario, parole OOV: *out of vocaboulary.*

Titolo	Author Index
Autore	
Origine	Very Large Data Bases
Anno	2000
Titolo	Author Index
Autore	?
Origine	VLDB
Anno	2000

Titolo	The active database management system manifesto: a rulebase of ADBMS features
Autore	Corporate Act-Net Consortium
Origine	ACM SIGMOD Record
Anno	1996
Titolo	ACT-NET - The Active Database Management System Manifesto: A Rulebase of ADBMS Features
Autore	?
Origine	SIGMOD Record
Anno	1996

4.3 Amazon Google

L'Amazon Google dataset[3] contiene descrizioni di diversi prodotti tratti da due siti di e-commerce rispettivamente Amazon e Google Products. Questo dataset contiene descrizioni più o meno lunghe che sporcano la rappresentazione più semplice, dunque ci si potrebbe aspettare risultati migliori dai modelli che includono reti ricorrenti. Si può notare un estratto in tabella 5.

In questo dataset vi sono 4589 tuple, e delle $\binom{4589+1}{2} = 10531755$ coppie possibili vi sono solamente 1300 *matches*. La percentuale dei *match* su tutte le coppie possibili è:

$$\frac{matches}{\binom{h+1}{2}} = \frac{1300 + 4589}{10531755} = 0.0559\%$$

Titolo	acad upgrade dragon naturallyspeaking pro solution 9.0 (a289a-fd7-9.0)
Descrizione	- marketing information: dragon naturallyspeakingprofessional 9 will saveyou and your organizationmoney because itempowers anyone tocreate documents overthree times faster thantyping. it also protectsagainst repetitive stressinjuries that result in lostproductivity higher workers compensationpremiums and higher temporary labor costs.the accuracy performance and ease-of-use indragon naturallyspeaking professional 9make it the ideal solution for busy corporateprofessionals. you and your employees canuse dragon naturallyspeaking professionalto create documents and email messages write reports and complete forms all by voice. product information - software sub type: voice recognition - software name: dragon naturallyspeaking v.9.0 professional - upgrade - features and benefits: - the most accurate ever - faster and safer than typing - section 508 certified - easy to use - no special user training required - support for citrix thin-client environments - enterprise ready - ease of administration - mobility - bluetooth support - robust customization - natural language capabilities - text to speech - language support: english - platform support: pc miscellaneous - compatibility: - microsoft office applications: word excel outlook andpowerpoint - corel wordperfect
Produttore	nuance academic
Prezzo	399.54
Titolo	edius pro 4
Descrizione	whether you are working with standard definition or high definition video grass valley edius pro nle software frees you from the limitations of conventional editing systems. edius pro software can be upgraded to edius broadcast at any time.
Produttore	canopus/grass valley
Prezzo	585.99

Tabella 5: Esempio di tuple nell'Amazon Google dataset

Col metodo utilizzato si utilizzano un numero totale di 17667 coppie.

Confrontato con gli altri due dataset utilizzati questo è più complesso, anche se con un numero di tuple complessivo comparabile al DBLP ACM, a causa dell'attributo descrizione.

Utilizzando t-SNE e PCA, utilizzate per i precedenti dataset, si da una rappresentazione bi-dimensionale dello spazio del dataset. Come si nota in figura 26 non vi è una chiara separazione tra i punti blu (coppie che non fanno *match*) ed i punti rossi (coppie che fanno *match*). É chiaro quindi che il dataset proposto è il più complesso tra quelli utilizzati. É quindi logico supporre che modelli più complessi, i quali possono codificare informazioni più complesse, possano rappresentare meglio questi dati e quindi classificare correttamente le coppie.

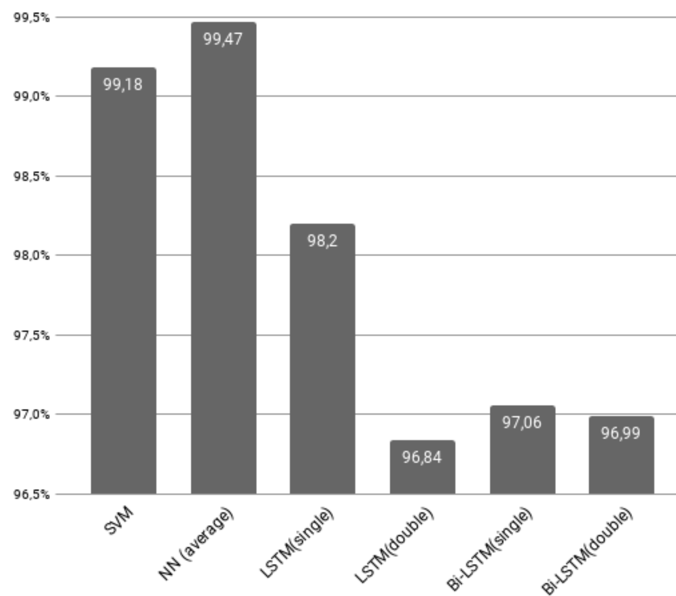
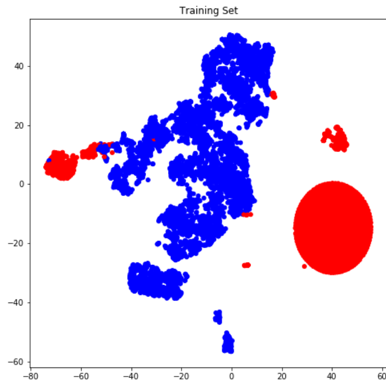


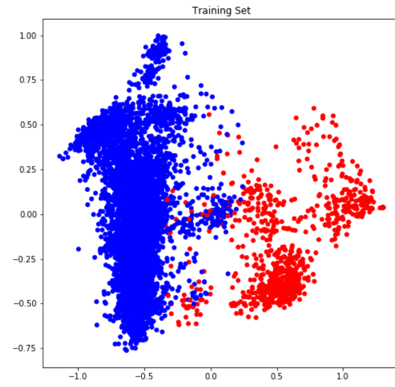
Figura 27: Amazon Google F_1-score

Come illustrato in figura 27 il modello che implementa una rete neurale illustrata nella sezione 15 che, a differenza delle precedenti e di come illustrato nella suddetta sezione, viene implementata con due strati nascosti composti da 8 unità cadauno, riesce al meglio a rappresentare le informazioni ed a separare più correttamente le diverse coppie.

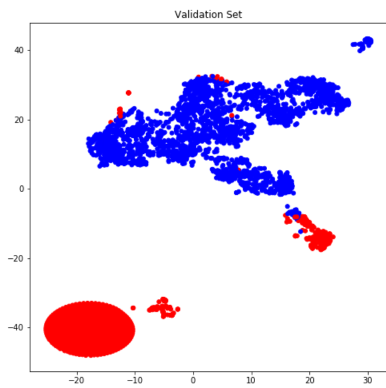
I modelli contenenti reti ricorrenti non riescono ancora a generalizzare meglio di quanto facciano i modelli che implementano la media. Inoltre, unitamente agli enor-



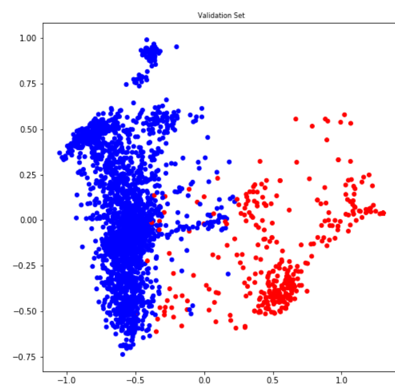
(a) Training Set t-SNE



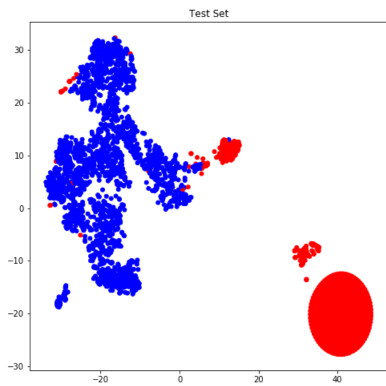
(b) Training Set PCA



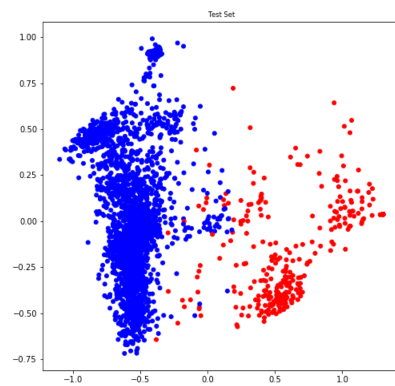
(c) Validation Set t-SNE



(d) Validation Set PCA



(e) Test Set t-SNE



(f) Test Set PCA

Figura 26: t-SNE applicata ad ognuna delle tre partizione del dataset

mi tempi di training, come illustrati nella figura 28, questi modelli risultano non competitivi in confronto alle SVM o alla rete neurale.

Valutando il test set rispetto al modello migliore sul validation set, quindi il modello NN con media si ottiene il seguente risultato:

	Precision	Recall	F_1
NN(average)	99.94%	99.29%	99.61%

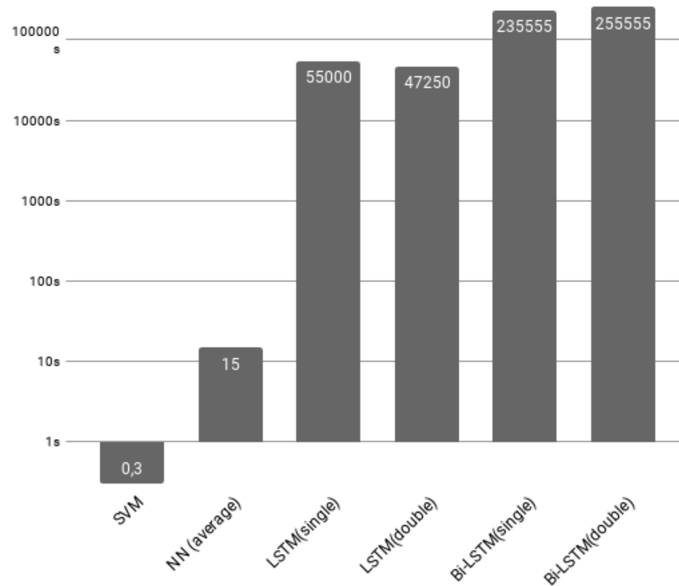


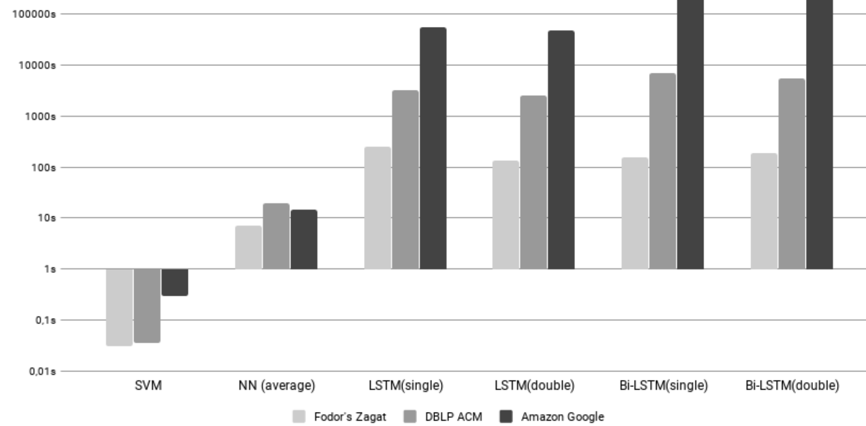
Figura 28: Amazon Google tempo di training in secondi su scala logaritmica

4.4 Riduzione del training set

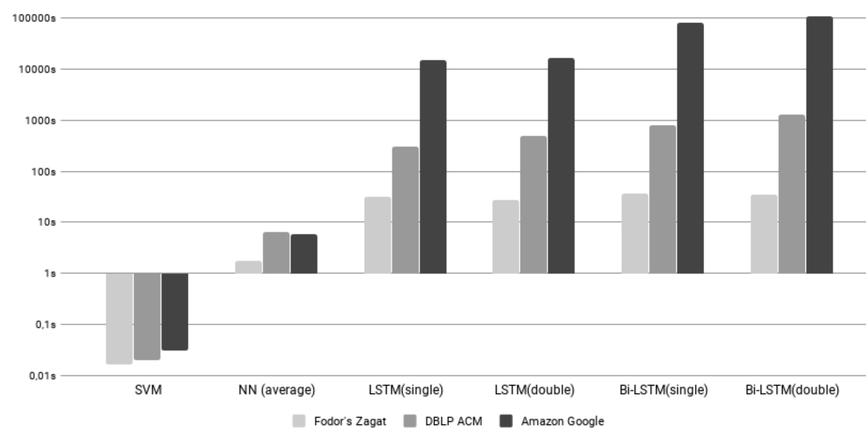
Potrebbe essere interessante valutare il comportamento ottenuto riducendo drasticamente la quantità di esempi forniti nel training set, simulando quindi una ridotta quantità di esempi conosciuti. Questo è ottenuto diminuendo la percentuale del training set a dispetto del validation e test set. Più precisamente usando le seguenti percentuali:

- *training set* 10% (rispetto a 60%);
- *validation set* 45% (rispetto a 20%);
- *test set* 45% (rispetto a 20%).

Questa importante riduzione riduce anche significativamente i tempi necessari ad ogni modello per imparare come separare le varie coppie come mostrato in figura 29.



(a) 60% training

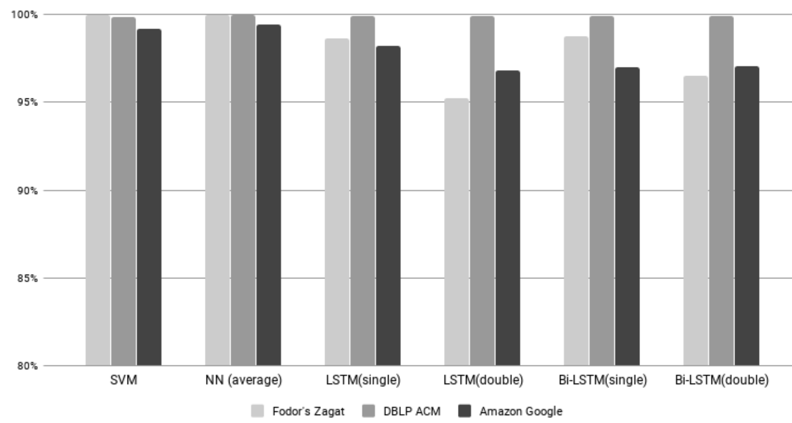


(b) 10% training

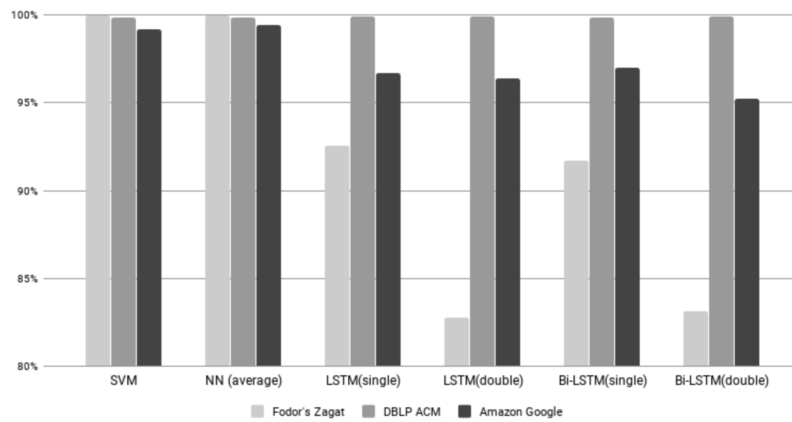
Figura 29: Confronto sul tempo in secondi su scala logaritmica

Riducendo drasticamente la dimensione del training set si può notare una complessiva e fisiologica diminuzione delle prestazioni, come illustrato in figura 30.

Si è passati per il Fodor's Zagat da 1756 tuple a 292, per il DBLP ACM da 12841 a 2140 e per l'Amazon Google da 11055 a 1842 tuple nell'insieme di *training*.



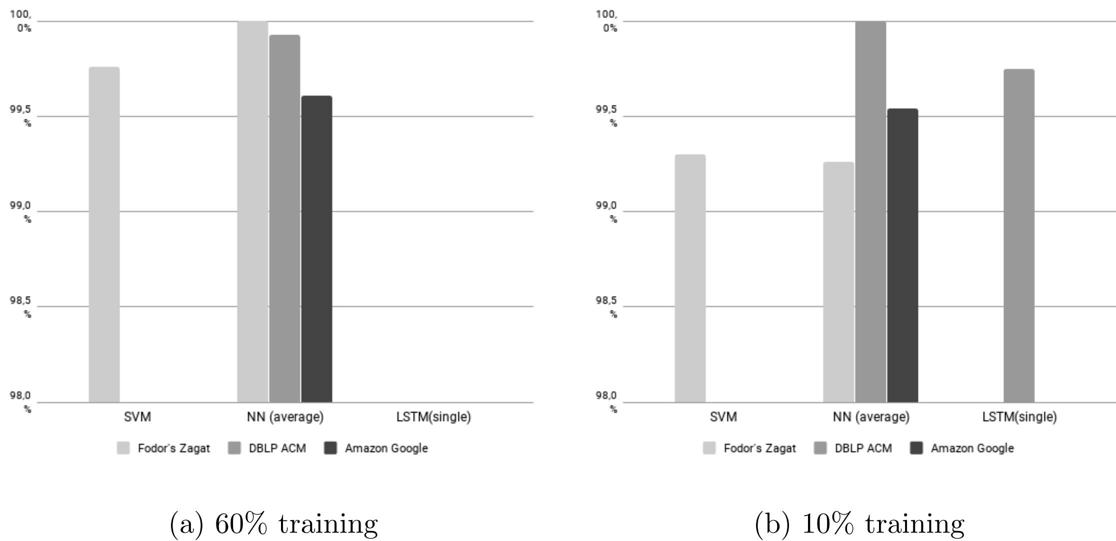
(a) 60% training



(b) 10% training

Figura 30: Confronto sul F_1

Si può notare inoltre che gli algoritmi che utilizzano una rappresentazione ottenuta con la media, quindi l'SVM e la NN(average), sono quelli che subiscono una flessione prestazionale complessiva inferiore sui vari dataset. Sono dunque gli algoritmi che, oltre ad essere i più performanti anche con un numero ridotto di esempi, sono anche quelli più resistenti.



(a) 60% training

(b) 10% training

Figura 31: Confronto sul F_1 del test set effettuato sugli algoritmi migliori

Anche i risultati sul *test set*, ottenuti esclusivamente sugli algoritmi migliori sul *validation set*, subiscono una riduzione, anche se di qualche decimo o centesimo di punto percentuale.

4.5 Considerazioni finali

Considerando i risultati mostrati nelle figure 29 e 30 è evidente come le soluzioni implementanti reti neurali ricorrenti per estrapolare informazioni: LSTM(single), LSTM(double), BiLSTM(single) e BiLSTM(double), ottengano in tempi molto lunghi scarsi risultati.

Gli algoritmi invece che implementano la media per rappresentare le tuple (vedi sezione 3.1.1): SVM e NN(Average), riescono ad ottenere risultati prossimi all'ottimo in tempi ragionevoli. La NN costruita come in figura 32, con un solo strato nascosto di 8 unità, o nel caso di dataset più complessi come l'Amazon Google due strati nascosti da 8 unità, ottimizzata con mini-batch gradient descent con learning rate $\eta = 0.01$ per dataset più semplici (Fodor's Zagat) e per dataset più complessi(DBLP ACM ed Amazon Google) anche con momento di nesterov $\alpha = 0.99$, riesce ad ottenere in tempi nell'ordine della decina di secondi il risultato migliore.

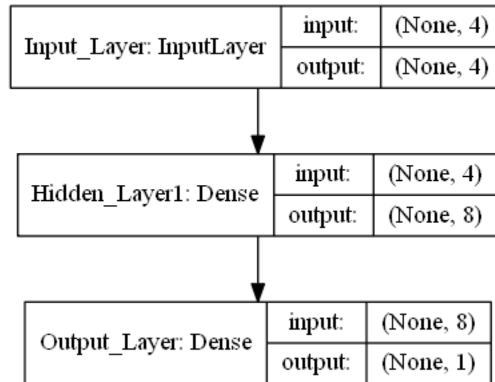


Figura 32: NN che ottiene i migliori risultati

Nel caso però si fosse interessati ad una implementazione che possa ottenere risultati prossimi a quelli migliori, con differenze di qualche decimo o centesimo di punto percentuale, in tempi molto ridotti, nell'ordine di decimi od addirittura centesimi di secondo (esclusa la parte di preparazione dei dati, che è uguale in entrambi), allora si può utilizzare la SVM fornita nella libreria scikit-learn [23] con kernel polinomiale di grado 2, la quale si è dimostrata la migliore scelta sui vari dataset.

Riferimenti bibliografici

[1] M. Ebraheem, S. Thirumuruganathan, S. R. Joty, M. Ouzzani, and N. Tang, "Deeper - deep entity resolution," *CoRR*, vol. abs/1710.00597, 2017. [Online]. Available: http://arxiv.org/abs/1710.00597

[2] S. Das, A. Doan, P. S. G. C., C. Gokhale, and P. Konda, "The magellan data repository," https://sites.google.com/site/anhaidgroup/projects/data.

[3] E. Rahm, H. Köpcke, and A. Thor, "Database group leipzig." [Online]. Available: https://dbs.uni-leipzig.de/en/research/projects/object_matching/fever/benchmark_datasets_for_entity_resolution

[4] G. Simonini, S. Bergamaschi, and H. V. Jagadish, "Blast: A loosely schema-aware meta-blocking approach for entity resolution," *Proc. VLDB Endow.*, vol. 9, no. 12, pp. 1173–1184, Aug. 2016. [Online]. Available: http://dx.doi.org/10.14778/2994509.2994533

[5] T. M. Mitchell, *Machine Learning*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997.

[6] D. Powers, "Evaluation: From precision, recall and f-factor to roc, informedness, markedness & correlation," vol. 2, 01 2008.

[7] Y. Sasaki, "The truth of the f-measure," 01 2007.

[8] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[9] N. Yadav, A. Yadav, and M. Kumar, *An introduction to neural network methods for differential equations*, ser. SpringerBriefs in Applied Sciences and Technology. Dordrecht: Springer, 2015. [Online]. Available: http://cds.cern.ch/record/1996681

[10] A. Ng, J. Ngiam, C. Y. Foo, Y. Mai, C. Suen, A. Coates, A. Maas, A. Hannun, B. Huval, T. Wang, and et al., "Optimization: Stochastic gradient descent." [Online]. Available: http://ufldl.stanford.edu/tutorial/supervised/OptimizationStochasticGradientDescent/

[11] G. Hinton, N. Srivastava, and K. Swersky, "The momentum method." [Online]. Available: https://www.coursera.org/lecture/neural-networks/the-momentum-method-Oya9a

[12] A. Ng, "Improving deep neural networks: Hyperparameter tuning, regularization and optimization." [Online]. Available: https://www.coursera.org/learn/deep-neural-network

[13] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[14] A. Ng, "Machine learning." [Online]. Available: https://www.coursera.org/learn/machine-learning

[15] ——, "Computer vision - foundations of convolutional neural networks." [Online]. Available: https://www.coursera.org/lecture/convolutional-neural-networks/computer-vision-Ob1nR

[16] ——, "Sequence models." [Online]. Available: https://www.coursera.org/learn/nlp-sequence-models

[17] J. Guo, "Backpropagation through time," 2013.

[18] S. Hochreiter and J. Schmidhuber, "Long short-term memory," vol. 9, pp. 1735–80, 12 1997.

[19] "Support vector machines." [Online]. Available: http://scikit-learn.org/stable/modules/svm.html

[20] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543. [Online]. Available: http://www.aclweb.org/anthology/D14-1162

[21] L. van der Maaten and G. Hinton, "Visualizing data using t-sne," 2008.

[22] A. Ng, "Principal components analysis." [Online]. Available: http://cs229.stanford.edu/notes/cs229-notes10.pdf

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[24] G. v. Rossum, "What is python? executive summary." [Online]. Available: https://www.python.org/doc/essays/blurb/

[25] ——, "Comparing python to other languages." [Online]. Available: https://www.python.org/doc/essays/comparisons/

[26] ——, "Foreword for "programming python" (1st ed.)." [Online]. Available: https://www.python.org/doc/essays/foreword/

[27] "Keras: The python deep learning library." [Online]. Available: https://keras.io/

A Ambiente

In questa appendice sono illustrate il linguaggio, le API, l'hardware attraverso i quali sono stati effettuati i test della sezione 4.

A.1 Specifiche

Seguono le specifiche della macchina sulla quale sono state raccolti i dati dei test effettuati:

CPU : AMD Ryzen 5 1600

Memoria : 8GB DDR4 2132MHz

A.2 Python

Python è un linguaggio di programmazione interpretato, orientato agli oggetti, di alto livello. Le sue strutture di alto livello insieme al *dynamic typing* e *dynamic binding*, lo rendono adatto per lo sviluppo rapido. [24]

Python nasce da Guido van Rossum partendo dall'ABC un linguaggio didattico monolitico che non prese piede negli anni '80. Python non è un linguaggio particolarmente rapido, ma permette di ridurre il tempo di sviluppo. Un programma in Python rispetto ad un programma Java esegue più lentamente, ma è tipicamente da 3 a cinque volte più corto. [25, 26]

Python è stato utilizzato principalmente in quanto l'API di Keras è scritta in Python.

A.3 Keras

Keras è un'API di alto livello per reti neurali scritta in Python e che può essere eseguita su Theano, CNTK, TensorFlow (quello utilizzato nei test). É stato sviluppato per permettere una veloce sperimentazione. Per andare dall'idea ai risultati col minor tempo possibile. [27]

Per tali motivazioni Keras è stato scelto ed utilizzato. Permette infatti di definire algoritmi, anche complessi in poche righe di codice (vedi appendice B). Permette anche training e valutazione in rispettivamente un'unica riga di codice.

B Codice

In questa sezione viene illustrata la struttura del codice e la sua motivazione ed inoltre vengono mostrati alcuni estratti del codice utilizzato per i test della sezione 4.

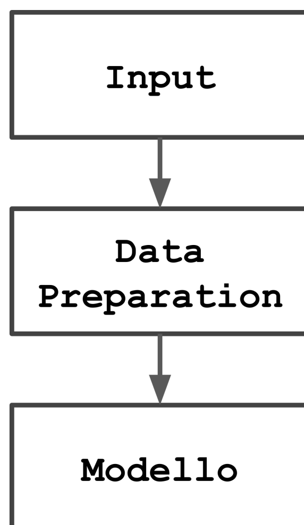


Figura 33: Struttura del codice

B.1 Struttura

La struttura del programma è a tre livelli che possono essere così definiti:

1. Input;
2. Data Preparation;
3. Machine Learning;

come illustrato in figura 33. Questa struttura può essere espansa siccome sono presenti input diversi e modelli diversi, come rappresentato in figura 34

B.2 Input

Input non è altro che una singola funzione il cui scopo è aprire il file od i file che contengono i vari dati e poi organizzarli in una forma ben precisa per il Data Preparation. Questa forma ben precisa consiste in due array bidimensionali *featureArray* e *paramArray*.

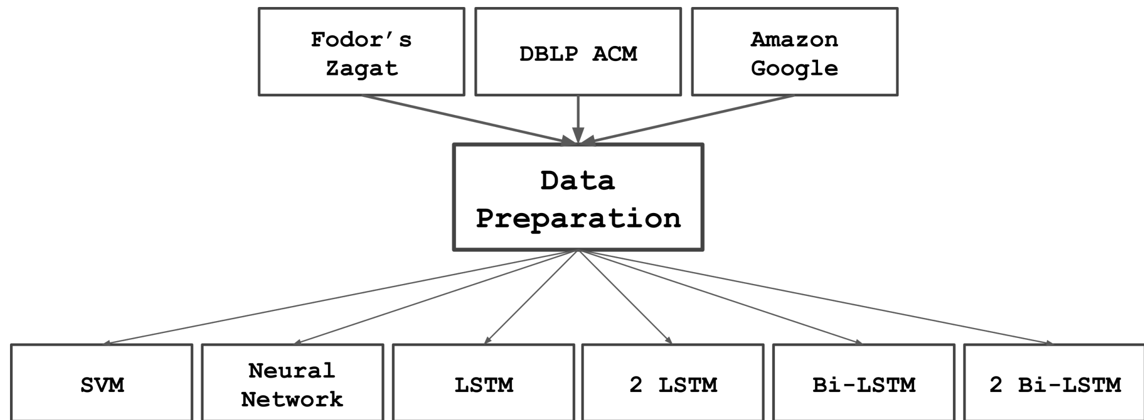


Figura 34: Espansione della struttura, sono mostrati i diversi input ed i diversi modelli

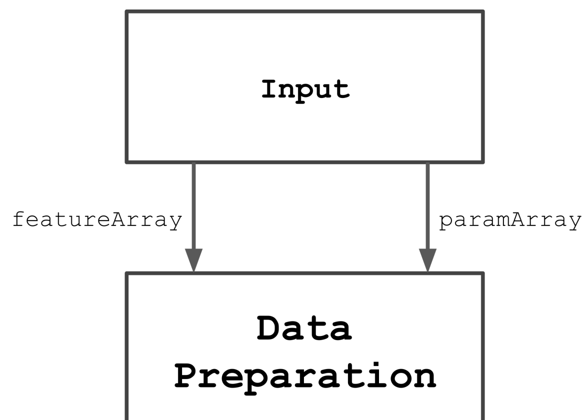
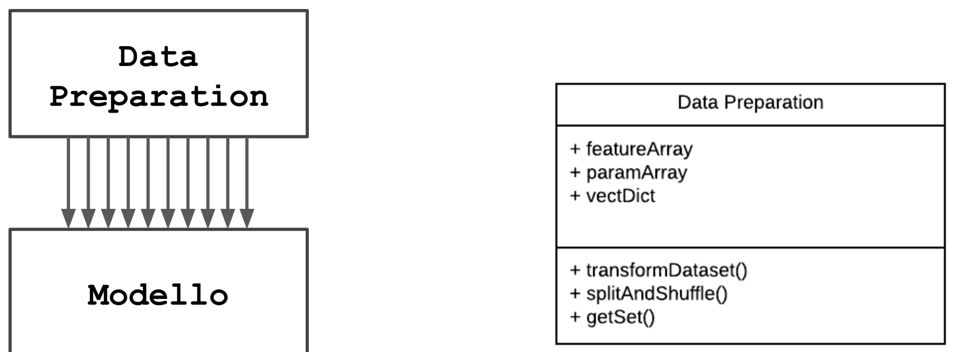


Figura 35: Interfaccia tra Input e Data Preparation

Il primo, *featureArray*, non è che una matrice che ha tante righe quante il numero di tuple e tante colonne quanto il numero di attributi, e contiene stringhe. Il secondo, paramArray, un array di dimensione uguale al numero complessivo di tuple, contiene numeri interi i quali rappresentano le singole entità, ovvero il numero all'indice j è uguale a quello all'indice k significa che le tuple i cui attributi sono contenuti agli indici j e k del featureArray fanno riferimento alla stessa entità reale e quindi fanno *match*.

B.3 Data Preparation

Il secondo livello è quello più complesso e ricco di funzioni. Esso prende i dati appena descritti e genera le informazioni necessarie al modello per compiere le sue funzioni.



(a) Valori presi dal Data Preparation necessari al modello

(b) UML

Figura 36: Interfaccia del Data Preparation e alcuni degli elementi più importanti rappresentati in UML

L'informazione più importante è un indice, un array bi-dimensionale, a cui viene fatto rifermento col nome di *tupleIndex* di forma $(h, 2)$ con h il numero complessivo delle coppie prese in esame costruite come indicato all'inizio della sezione 4, e quindi un totale di $3(\textit{numero di match})$. Ogni riga contiene una coppia di indici i quali fanno riferimento a due tuple descritte dal *featureArray* e *paramArray*. Questo vettore, ottenuto tramite la funzione *transformDataset()*, ha per esempio questa forma:

$$\begin{bmatrix} 1 & 3 \\ 154 & 234 \\ \vdots & \vdots \\ 0 & 56 \end{bmatrix}$$

La funzione *splitAndShuffle()* ha lo scopo di creare tre array di interi: *trainingIndexs*, *valIndexs*, *testIndexs*, che rappresentano indici del *tupleIndex* e quindi costituiscono tre insiemi di coppie di tuple. Questi tre insiemi disgiunti formano i tre insiemi: training, validation e test visti nella sezione 4.

Per ultima, la funzione *getSet()* fornisce la corretta rappresentazione richiesta dal modello dell'insieme di indici fornito (*trainingIndexs*, *valIndexs* o *testIndexs*).

B.4 Modello

L'ultimo strato, il modello, contiene le funzioni di Keras per creare il modello, insegnarli tramite l'insieme di training, e valutarlo su un dato insieme. Tutte queste funzioni sono raggruppate in poche linee di codice come illustrato di seguito.

```python
model=Sequential()

model.add(InputLayer(input_shape=(4,),name="Input_Layer"))
model.add(Dense(units=8, activation='tanh',kernel_initializer=
    initializers.glorot_uniform(seed=1234),name="Hidden_Layer1"))
model.add(Dense(units=1, activation='sigmoid', kernel_initializer=
    initializers.glorot_uniform(seed=4321),name="Output_Layer"))

model.compile(loss='binary_crossentropy', optimizer=SGD(momentum=.99,
    nesterov=True), metrics=['accuracy',precision,recall])
```

Listing 1: Creazione del modello

```python
model.fit(trainSetX, trainSetY,validation_data=(valSetX,valSetY), epochs
    =200, batch_size=256)
```

Listing 2: Insegnamento

```python
metrics = model.evaluate(testSetX, testSetY, verbose=0)
```

Listing 3: Valutazione