Entity Matcher AirBnB

Pier Vincenzo De Lellis PIE.DELELLIS@STUD.UNIROMA3.IT
Dipartimento di Ingegneria Informatica
Università degli studi Roma Tre

Francesco Foresi FRA.FORESI@STUD.UNIROMA3.IT
Dipartimento di Ingegneria Informatica
Università degli studi Roma Tre

Andrea Giorgi AND.GIORGI4@STUD.UNIROMA3.IT
Dipartimento di Ingegneria Informatica
Università degli studi Roma Tre

Progetto: Analisi e Gestione dell'Informazione sul Web, 2020/21

Abstract

L'Entity Resolution (ER) è un task fondamentale nel contesto dell'integrazione dei dati, in particolare l'ER si occupa di rilevare se dati due record da sorgenti differenti, questi si riferiscano alla stessa entità. A causa della sua complessità intrinsecamente quadratica, si è cercato per anni una serie di tecniche per accelerare la risoluzione di questo task per giungere a un'adeguata riduzione di dati voluminosi. Nonostante gli sforzi e i buoni risultati ottenuti dal 1969, utilizzando per 30 anni tecniche rule-based and stats-based, soltanto nel 2000 si è pensato di utilizzare un approccio basato sul machine learning, utilizzando Support Vector Machine SVM o Decision Tree per effettuare il matching tra i records, ottenendo F-Score compresi tra il 70 e il 90 percento. Solo nel 2018 viene utilizzato per la prima volta la tecnica del Deep Learning e dell'Entity Embedding. Vi è ancora una forte richiesta di democratizzazione dell'ER, riducendo il pesante coinvolgimento umano nell'etichettatura dei dati, eseguendo feature engineering e ottimizzando i parametri. In questa relazione viene analizzato l'efficacia dell'approccio basato sul Deep Learning sul dataset fornito da AirBnB, utilizzando due delle reti neurali più discusse nello stato dell'arte, DeepMatcher e DeepER.

Keywords: Entity Resolution, Entity Matching, Deep Learning

Link gitHub: https://github.com/DFG-team/EM-airBnB

1. Introduzione

L'Entity Resolution ER è il processo di creazione di un matching tra diversi record riguardanti una stessa entità in assenza di una chiave di join. Nel caso d'uso in considerazione, l'entità sulla quale facciamo riferimento è una casa registrata su AirBnB e il task che si vuole risolvere è individuare se due annunci differenti facciano riferimento alla stessa abitazione. Per semplicità, si considerino due annunci riguardanti stanze diverse della stessa casa come fossero la stessa abitazione. L'ER ha portato grandi vantaggi nello sviluppo dell'integrazione

©2021 Pier Vincenzo De Lellis, Francesco Foresi e Andrea Giorgi.

dei dati, questo perché un essere umano è in grado di fornire un accuracy del matching pari al 100%, ma non è grado di elaborare una grande mole di dati. Infatti, se considerassimo già 1000 records per entrambi i dataset allora ci sarebbero 1 milione di coppie da controllare; la complessità aumenta quindi rapidamente con dataset sempre più grandi. I recenti sviluppi ottenuti negli ultimi anni sul Machine Learning e più nello specifico sul Deep Learning, hanno generato un enorme impatto anche sulle tecniche circa l'Entity Resolution (ER). In particolare si è visto come sia possibile addestrare un classificatore binario per risolvere il suddetto problema, avendo un dataset etichettato con 0 o 1 (1 se i record della coppia si riferiscono alla stessa casa, 0 altrimenti). In questo elaborato, viene discussa l'applicazione di due metodi di Deep Learning dello stato dell'arte DeepMatcher e DeepER sul dataset fornito da AirBnB. A seguito di una fase di pre-processamento del suddetto dataset, sono state addestrate le due reti neurali con i dati di training relativi agli annunci della città di Roma. Infine, i modelli ottenuti dall'addestramento sono stati testati sulle seguenti città Europee: Amsterdam, Londra, Dublino, Bruxelles.

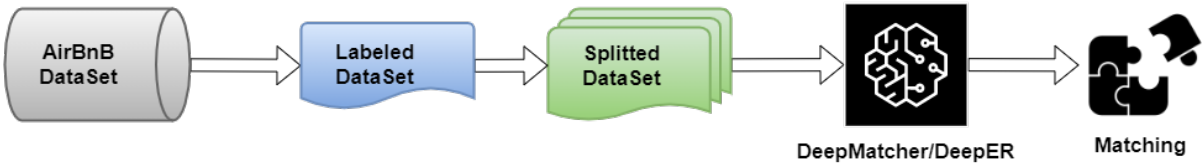

Figure 1: Pipeline di sperimentazione DeepMatcher/DeepER.

2. Pre-processamento dataset

Nella sperimentazione svolta, la pipeline di esecuzione inizia con una fase di pre-processamento del dataset. Sono stati utilizzati per l'addestramento i dati, forniti da *insideairbnb.com*, relativi alle case situate a Roma, per poi testare i modelli sui dati di *Londra*, *Amsterdam*, *Dublino* e *Bruxelles*. AirBnB fornisce i listing di case -oggetto di annuncio sull'omonimo sito- in diverse forme, tra cui:

- Listing dettagliato di dati;
- Listing dettagliato di dati di calendario;
- Listing dettagliato di dati di recensioni;
- Listing di informazione riassuntive e di metriche.

Utilizzare listing dettagliati sarebbe la casistica ottimale, in quanto in grado di ricavare informazioni utili per il matching delle entità anche da dati non strettamente collegate all'entità stessa (*e.g:* bio dell'host, recensioni, ecc..). Vi sono però dei problemi rilevanti collegati all'utilizzo di questi listing nella sperimentazione di sistemi come *DeepMatcher* e *DeepER*. In particolare, addestrare i modelli su questo tipo di dataset necessita la configurazione di *162,759,876* parametri, con un numero di attributi per coppia di identità pari a *150*, richiedendo una potenza di calcolo troppo elevata per l'hardware utilizzato in questa sperimentazione. Problema duale è stato riscontrato utilizzando come input dati di *ground*

truth creati con approccio *rule-based* riguardanti le case situate a Roma. Questo listing, oltre ad essere poco recente rispetto ai dati forniti da AirBnB (aggiornati a Dicembre 2020), contiene una mole di dati poco significativa (ca *300* coppie etichettate ad 1) ed ogni casa è rappresentata da pochi attributi rispetto ai listing standard, dunque necessiterebbe di un ulteriore pre-processamento. Inoltre utilizzare un listing del genere risolve il problema dell'etichettatura solo per gli esempi positivi, dovendo in ogni caso pre-processare i dati di AirBnB per generare coppie etichettate a 0. Si è optato quindi per l'utilizzo di listing di informazioni riassuntive che, anche se più snelli rispetto ai listing dettagliati, permettono il raggiungimento di prestazioni decisamente soddisfacenti. Poste dunque tali premesse, nelle prossime sotto sezioni si definiscono nel dettaglio le fasi della pipeline di sperimentazione relative al pre-processamento del dataset, ossia lo split e il labeling.

2.1 Labeling

Il labeling è una fase necessaria per l'utilizzo dei due modelli, infatti entrambi richiedono come input un listing di un tipo specifico per poter processare il data frame. In particolare ogni record del listing (*i.e.* coppia di annunci) necessita una colonna per il *label* (1 se gli annunci riguardano la stessa abitazione, 0 altrimenti) e una per l'*id*. Sono stati implementati tre metodi, due per etichettare con 0 (no-match), uno per etichettare con 1 (match):

- Le coppie di case etichettate con 1 sono ottenute facendo un merge del dataset con se stesso, con *join* sulle colonne *hostId*, *latitude* e *longitude*.
- Le coppie di case etichettate con 0 sono di due tipi, uno ottenuto facendo un merge del dataset con se stesso con *join* sulle colonne *latitude*, *roomType* e *price*, l'altro con join invece sulle colonne *longitude*, *neighbourhood* e *minimum nights*.

In entrambi i casi di etichettatura 0, i risultati del merge sono sottoposti a controllo che i membri delle coppie prodotte dal join abbiano *hostId*, *latitude* e *longitude* diversi tra loro, così da verificare che non facciano già parte di esempi etichettati con 1. Una volta ottenute le coppie etichettate con match viene eseguita una raffinazione manuale per eliminare gli annunci non classificati correttamente. Successivamente, si aggiunge la colonna *id* per permettere alla rete neurale di identificare le coppie di annunci ricevuti come input con un identificativo univoco. La fase di labeling però varia a seconda del modello: infatti, esclusi id e label, DeepMatcher richiede di aggiungere agli attributi delle due coppie i prefissi *left* per gli attributi riguardanti il primo annuncio e *right* per gli attributi del secondo annuncio. Analogamente, DeepER richiede i prefissi *ltable* e *rtable* ai rispettivi annunci delle abitazioni presi da due sorgenti distinte.

2.2 Splitting

Nella seconda fase del pre-processamento, sono scelti i dati riguardanti la città di Roma per effettuare l'addestramento su entrambi i modelli. Il dataset è suddiviso in 3 parti:

- Training set, 60% del dataset;
- Validation set, 20% del dataset;
- Test set, 20% del dataset.

Le percentuali delle suddivisioni sono scelte in modo da massimizzare i dati ottenuti dal dataset, numpy offre di default questa configurazione. Una volta effettuata la suddivisione, sono salvati i tre nuovi file in formato *csv*, passati come input ad entrambe le reti neurali per iniziare la fase di addestramento. Per la fase di test, si utilizzano i listing per intero di *Amsterdam*, *Londra*, *Dublino*, *Bruxelles*. Essendo dataset più piccoli, lo split rimane comunque bilanciato rispetto al training set e al validation set: considerando l'intero dataset di Roma contenete circa 28,000 record, dopo averlo suddiviso come descritto precedentemente, vi sarà un equo bilanciamento tra training set e test set. L'utilizzo di entrambi i tipi di etichettatura con 0 però è da pesare in maniera adeguata, in quanto causa uno squilibrio tra esempi labeled 0 e esempi labeled 1, influenzando l'*anomaly detection* e riportando bassi valori di *recall*. In effetti, questo problema risulta più evidente nel testing di *DeepER*, più sensibile a questo squilibrio tra esempi negativi e positivi, dunque si è esclusa la doppia etichettatura negativa. Mentre per quanto riguarda *DeepMatcher*, è stato possibile includere entrambi i tipi di label 0; ad eccezione di Londra e Amsterdam, per cui si genera un test set di circa 30,000 record. Si è optato dunque per l'utilizzo di un solo tipo di etichetta negativa per il bilanciamento dei dataset delle città di Amsterdam e Londra, mentre per il resto delle città è stato possibile mantenere di default entrambi i label 0.

3. Modelli

La fase sperimentale si è focalizzata sull'analisi e l'esecuzione di due soluzione proposte per il problema dell'Entity Matching EM. Il processo di Entity Matching compiuto da entrambi i modelli può essere definito formalmente da Mudgal et al. (2018) come segue: Definiamo come *entity* un oggetto reale e definiamo come *entity mention* un riferimento verso un entità del mondo reale. Dunque sia D e D' due collezioni di entity mention, assumiamo che gli elementi nelle collezioni seguano la stessa rappresentazione. L'obiettivo dell'Entity Matching EM è quello di trovare tutte le coppie di *entity mention* tra D e D' che fanno riferimento alla stesso oggetto reale. Le coppie individuate vengono definite come match e questo processo viene effettuato in due fasi: blocking e matching. La fase di blocking filtra il prodotto incrociato $D \times D'$ producendo un set candidato C contenente solo le coppie di *entity mention* che probabilmente sono dei match. La fase successiva è quella di matching, la quale prende in input le collezioni D e D' assieme al set candidato C. Il matching sfrutta anche un set di tuple T contenente le coppie di entity mention e un valore di label l che può prendere due valori {"match", "no-match"}. Dato il set T di dati etichettati l'obiettivo di un matcher è quello di distinguere tra coppie "match" e coppie "no-match", che nella nostra implementazione sono state etichettate con {"1", "0"} nella fase di labeling.

3.1 DeepMatcher

Mudgal et al. (2018) introduce un soluzione notevole per il problema di matching basata sull'utilizzo di diverse tecniche di Machine Learning ML, nello specifico le soluzioni proposte sfruttano le potenzialità del Deep Learning DL applicate al campo della Natural Language Processing NLP. Le soluzioni sono state progettate ed implementate seguendo la stessa architettura di supporto. Questa architettura è composta da tre macro-blocchi, ciascuno specializzato in un task notevole per il processo di Entity Matching:

- *Attribute Embedding Module*: per ciascun attributo A_j appartenente all'entity e_1 o e_2 prende le sequenze di parole relative alle entity mention e_1 ed e_2 e le converte in due sequenze di vettori di embedding $u_{e_1,j}$ e $u_{e_2,j}$ i cui elementi corrispondono ad un embedding d-dimensionale delle parole corrispondenti. L'output sarà una coppia di embedding $u_{e_1,j}$ e $u_{e_2,j}$ per i valori degli attributi A_j per le entity mention e_1 ed e_2;
- *Attribute Similarity Representation Module*: il suo obiettivo è quello di apprendere automaticamente una rappresentazione per catturare la similarità tra gli attributi relativi agli embedding u_j di due entity mention, i quali vengono passati come input. Le operazioni compiute in questo modulo sono suddivisibili in due fasi: (1) Attribute summarization, fase che applica un operazione H alle due sequenze in input per poterle riassumere, aggregando le informazioni in tutti i token nella sequenza dei valori degli attributi dell'entity mention, e (2) Attribute comparision, la quale prende come input i vettori risultanti dalla fase di summarization e produce un valore di similarità totale tra e_1 ed e_2 applicando una funzione $D(s_{e_1,j}, s_{e_2,j})$ dove $s_{e_1,j}$ rappresenta la summarization di $u_{e_1,j}$;
- *Classification Module*, prende in input le rappresentazioni di similarità e le usa come feature per un classificatore M che determina se le entity mention in input fanno riferimento allo stesso oggetto reale.

Sulla base di questa architettura sono state realizzate quattro implementazioni diverse, ciascuna caratterizzata dal proprio approccio risolutivo al problema; tutte le quattro soluzioni utilizzano fastText, un embedder pre-addestrato che viene implementato come rappresentante del modulo di Attribute Embedding. Inoltre, tutte le quattro soluzioni sfruttano una rete neurale ReLU HighwayNet, una rete caratterizzata da tempi di convergenza ridotti e da una capacità migliore di produrre risultati empirici rispetto ad una rete neurale tradizionale, questa è seguita da un layer Softmax per implementare il modulo di Classification. L'unico modulo che differisce nelle quattro soluzioni è il modulo di Attribute Summarization.

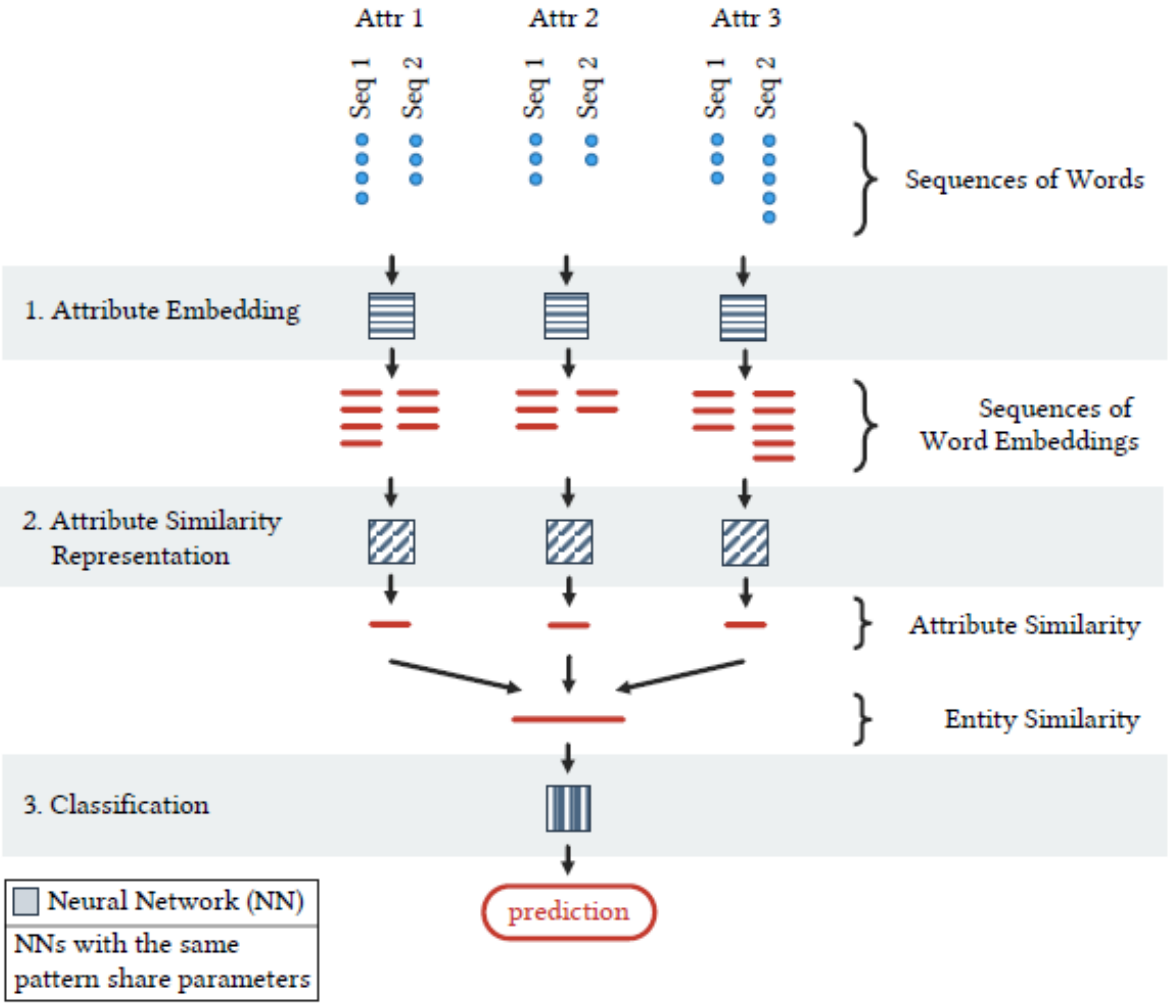

Figure 2: DeepMatcher: Architettura proposta da Mudgal et al. (2018).

3.1.1 SIF: Aggregate Function Model

Questo modello è caratterizzato da una funzione aggregata per produrre l'input per il modulo di classificazione. La funzione aggregata è composta da una funzione di media pesata per l'attribute summarization e una funzione di element-wise absolute difference per l'operazione di comparazione. La funzione di averaging segue il modello Sentence Inverse Frequency SIF di embedding delle frasi introdotto da Arora. Le prestazioni del modello fanno riferimento principalmente al potere espressivo della fase di embedding e dal classificatore adoperato.

3.1.2 RNN: Sequence-aware Model

Questo modello sfrutta una RNN bidirezionale per la fase di attribute summarization e per quella di element-wise absolute difference. Questo modello tiene conto dell'ordine delle

parole presentate in input. Il modello RNN di riferimento è quello di una GRU-based RNN bidirezionale, ed è composto da due RNN: (1) Una forward RNN che processa in ordine la sequenza di word embedding u per produrre gli hidden states $f_{1:t}$ e (2) una rete backward RNN che processa la sequenza di input nell'ordine inverso per produrre gli hidden states $b_{t:1}$. La fase di attribute summarization concatenerà gli ultimi due output della RNN bidirezionale.

3.1.3 Attention: Sequence Alignment Model

Questo modello sfrutta una decomposable attention per implementare la attribute summarization e una vector concatenation per compiere la attribute comparision. Analizza entrambe le sequenze in input imparando nel mentre la rappresentazione di similarità. Per computare la summarized representation per due sequenze u_1 e u_2, si sfrutta u_1 come input primario e u_2 come contesto per l'attention model. Sono tre gli step: (1) Soft Alignment, a partire da una soft alignment matrix W, i cui valori rappresentano i pesi ottenuti dalle hidden rappresentation della HighwayNet, per una coppia di elementi $u_1[k]$ e $u_2[m]$. Viene computato l'encoding $b_1[k]$ per ogni $u_1[k]$ utilizzando la media pesata di tutti gli elementi $u_2[m]$, con i pesi provenienti dalla k-esima riga della matrice W. (2) Comparision, vengono confrontati gli embedding $u_1[k]$ con il soft-aligned encoding $b_1[k]$ usando una HighwayNet, denotando con $x_1[k]$ la comparision representation. (3) Aggregation, si sommano tutti i vari comparision representation di u_1 e si normalizza la somma per la radice quadrata del modulo di u_1. Questi step vengono ripetuti usando u_2 come input primario e u_1 come contesto.

3.1.4 Hybrid Model

Questo modello rappresenta l'unione tra una RNN bidirezionale e di una soluzione basata sulla decomposable attention per implementare un attribute summarization e una vector concatenation migliorata tramite una element-wise absolute difference durante la fase di attribute comparision. Il modello si basa su tre step fondamentali: (1) Soft Alignment, viene costruita la matrice W tra l'input primario u_1 e il contesto u_2 e vengono determinati i soft-aligned encoding $b_1[k]$ utilizzando una media pesata sugli enconding di u_2, ottenuti concatenando gli hidden states di una RNN_1 bidirezionale che prende come input u_2. (2) Comparision, la comparazione tra b_1 e l'input primario u_1 prevede di usare $BI - RNN_1$ creando gli encoding di u_1 producendo u_1' e si procede con una element-wise comparision tra b_1 e u_1'. Si usa una HighwayNet per produrre la comparision representation $x_1[k]$ per ciascun u_1. (3) Aggregation, si aggregano i valori di x_1 usando uno schema apposito di calcolo dei pesi. Il calcolo sfrutta una $BI - RNN_2$ che determina gli encoding g_2 di u_2, si concatenano i singoli $x_1[k]$ con g_2 passandoli ad una HighwayNet seguita da un layer Softmax. Come ultimo step si compie una media pesata su tutti gli elementi di x_1 usando questi pesi.

3.1.5 Soluzione Proposta

Nella fase sperimentale descritta da Mudgal et al. (2018) vengono paragonate le prestazioni dei quattro modelli proposti rispetto alle prestazioni di Magellan, una soluzione notevole per il problema di Entity Matching EM. Vengono descritti quattro tipologie di problemi: (1) dati strutturati, (2) dati testuali, (3) dati testuali con attributi, (4) dati sporchi con errori e vengono paragonate le prestazioni dei modelli rispetto a questi quattro casi d'uso. Si nota come i modelli proposti, sopratutto l'Hybrid Model, sorpassano le prestazioni di Magellan ponendosi come una valida alternativa per la risoluzione di questo problema. Nella fase implementativa e sperimentale eseguita abbiamo scelto di adoperare l'Hybrid Model, il quale presenta le prestazioni migliori rispetto al nostro dominio applicativo.

3.2 DeepER

Ebraheem et al. (2018) introduce una soluzione alternativa e valida nel campo dell'entity resolution, che insieme al modello *DeepMatcher* proposto sempre nello stesso anno da Mudgal et al. (2018), sarà utilizzata ampiamente e con ottimi risultati. Infatti, solo recentemente sono stati sviluppati dei modelli migliori tra cui BERTA, EMT e DITTO. DeepER è un sistema che cerca di perseguire la democratizzazione dell'ER ed ha bisogno di molti meno dati etichettati rispetto alle classiche soluzioni, inoltre cattura le somiglianze sintattiche e semantiche senza l'utilizzo di ingegnerizzazione dei dati o regolazione dei parametri. Infine la soluzione proposta sfrutta potenzialità del Deep Learning DL applicate al campo della Natural Language Processing NLP. Infatti sono stati utilizzati metodi noti come *word2vec* e *GloVe* poiché sono in grado di catturare la somiglianza semantica tra stringhe. Inoltre, utilizzando embeddings di parole pre-addestrate (GloVe è addestrato su 840 milioni di token), possiamo ridurre enormemente l'intervento umano di etichettatura dei valori corrispondenti ad un set di dati. La soluzione proposta in DeepER presenta l'architettura mostrata in figura 3. Nel dettaglio sono analizzati gli elementi dell'architettura come segue:

- *Embedding e Composition Layers*: dato un attributo, per ogni elemento dell'attributo si genera un token utilizzando un tokenizzatore standard. Per ogni token (parola) x, consultiamo il dizionario GloVe precedentemente addestrato e ricaviamo un vettore v d-dimensionale. Se una parola non è trovata nel dizionario di GloVe o l'attributo ha un valore nullo, GloVe assegna un token special *UNK* per rappresentare parole fuori dal vocabolario (anche agli id viene assegnato il token speciale *UNK*). Infine la rappresentazione vettoriale di un attributo viene ottenuto mediante l'utilizzo di una RNN composta da celle LSTM, con memoria a breve e a lungo termine.

- *Similiarity Layer*: una volta ottenuta una *DR* (distribuited rappresentation) per entrambe le tuple, il prossimo step è calcolare la similiarità tra di esse. Si evidenzia che la DR ottenuta dall'averaging è un vettore con $d \times m$ dimensioni, a seguito di ciò applichiamo la *cosine similiarity* su tutte le d dimensioni (ogni dimensione corrisponde ad un attributo) ottenendo un vettore m-dimensionale. Al contempo, per le DR calcolate con le LSTM utilizziamo la vector difference e il prodotto *hadamard*, ottenendo un vettore x-dimensionale.

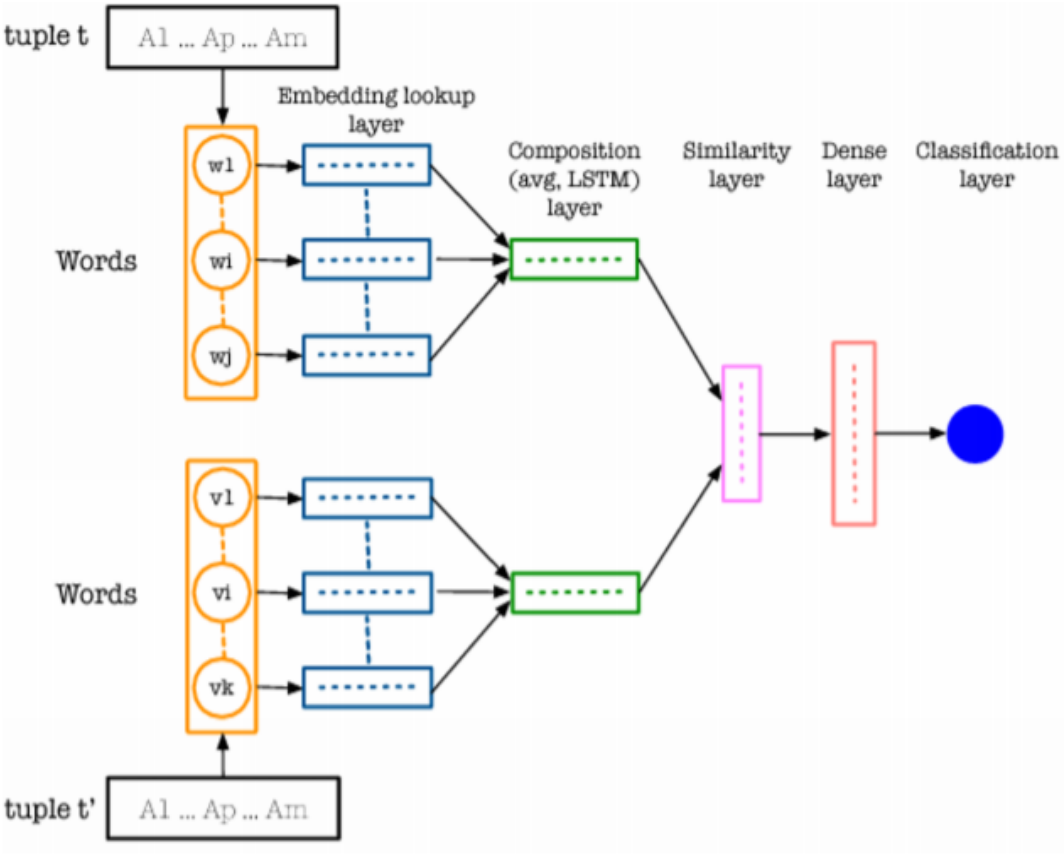

Figure 3: DeepER: Architettura proposta da Ebraheem et al. (2018)

- *Classification Layer*: riceve come input un Tabella T , il training set S, e un iperparametro L. Si utilizza un classificatore ER con *LSH* (Local Sensitive Hashing) basato su Blocking. Nel dettaglio, si generano delle funzioni hash usando il *random hyperplane method*. Successivamente per ogni tupla t, si indicizza la DR di t dentro L tabelle hash. Per ogni coppia di tuple di ogni tabella hash si applica il classificatore su (t, t').

L'ultimo strato dell'architettura (classification layer) fornisce come output il risultato del matching su tutte le tuple passate in input, 1 se le due tuple si riferiscono alla stessa entità, 0 altrimenti.

4. Sperimentazione

La sperimentazione dei due modelli è avvenuto in ambienti differenti a seconda del modello in questione. In particolare *DeepER* è stato testato in locale, eseguendo l'intera pipeline con l'IDE *PyCharm*, mentre per *DeepMatcher* il pre-processamento del dataset è avvenuto in

locale come per *DeepER*, mentre per l'esecuzione dell'addestramento del modello si è optato per la piattaforma cloud *CoLab*, che fornisce un massimo di 12GB di RAM gratuiti su cloud. L'hardware utilizzato per la sperimentazione è un DELL Inspiron 15 7000 con 16GB di RAM, processore Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz, scheda grafica NVIDIA GeForce GTX 1060Ti. Per l'esecuzione dell'addestramento è stato possibile l'impiego di Tensorflow con GPU, con CUDA 11.2 e cuDNN 8.1.10.

4.1 Risultati: DeepMatcher

Complessivamente i risultati ottenuti con DeepMatcher sono stati migliori rispetto a DeepER, a scapito però del tempo di esecuzione. Infatti per eseguire l'addestramento sulla piattaforma *CoLab*, la preparazione dell'ambiente, l'addestramento e il testing hanno impiegato circa 30 minuti per ogni esecuzione. In particolare sono stati riscontrati problemi circa la corretta importazione di *fasttextmirror* (una libreria per l'apprendimento efficiente delle rappresentazioni di parole e della classificazione delle frasi). In una successiva iterazione si potrebbe pensare infatti di eseguire anche DeepMatcher in locale con l'IDE *PyCharm*, in modo da velocizzare i tempi di addestramento e di testing. Nella tabella sottostante sono riportate le metriche ottenute nella fase di test e la grandezza del dataset utilizzato.

	Precision	**Recall**	**F-score**	**Test set**
Londra	0.991	1.000	0.995	9,529
Roma	0.991	1.000	0.995	4,154
Bruxelles	0.972	1.000	0.985	2,203
Amsterdam	0.973	1.000	0.986	1,659
Dublino	0.977	1.000	0.988	1,469

Table 1: Risultati ottenuti con DeepMatcher

Si noti come è stato possibile utilizzare due tipi di etichettatura 0, essendo il modello poco sensibile a questo sbilanciamento verso gli esempi con label negativo. Per quanto riguarda Londra ed Amsterdam, per evitare una crescita significativa del test set si è optato per un singolo label negativo.

4.2 Risultati: DeepER

Il modello DeepER ha fornito ottimi risultati sopratutto in relazione al tempo di esecuzione e all'ottimizzazione fornita dagli sviluppatori. Infatti, si nota come utilizzando la configurazione tensorflow con GPU, come riportato nella sottosezione precedente, il tempo di esecuzione per ogni epoch non supera mai i 10 secondi, anche con grandi moli di dati. In generale il tempo di esecuzione complessivo, considerando la preparazione dell'ambiente, il training e il testing, è di circa 5 minuti. Ciò è dovuto anche all'utilizzo della best practice Early Stopping per l'addestramento della rete neurale: dopo 5/7 epoch in cui l'accuracy dell'addestramento rimane invariata o peggiora, il sistema termina l'addestramento e procede direttamente con la fase di testing. Complessivamente quindi DeepER si è rilevato

un ottimo modello che offre un compromesso tra efficacia ed efficienza. Nella tabella sottostante sono riportate le metriche ottenute nella fase di test e la grandezza del dataset utilizzato.

	Precision	**Recall**	**F-Score**	**Test set**
Londra	0.513	0.988	0.676	9,529
Amsterdam	0.725	0.966	0.828	1,659
Bruxelles	0.974	0.900	0.937	1,600
Roma	0.981	0.979	0.980	1,125
Dublino	0.912	0.986	0.947	493

Table 2: Risultati ottenuti con DeepER

Si noti come la bontà della predizione sia inversamente proporzionale alla grandezza del test set, vi è dunque un comportamento peggiore per quanto riguarda un numero elevato di annunci, anche se l'addestramento è svolto su un dataset grande come quello di Roma. Inoltre si evidenzia il fatto che per il dataset di Roma vi sono annunci sia in lingua italiana che in lingua inglese, a differenza di tutti gli altri dataset presentanti esclusivamente annunci in lingua inglese; ciò può influenzare negativamente la bontà della predizione. Infine, come già anticipato, non è stato possibile utilizzare entrambi i tipi di label negativi, per via della troppa sensibilità del modello allo squilibrio tra numero degli esempi positivi e quelli negativi.

5. Conclusioni e Sviluppi Futuri

In questo elaborato è stata presentata l'implementazione e l'analisi dei modelli DeepMatcher e DeepER sul problema dell'Entity Resolution. Il problema che affronta ER è quello di verificare che due record provenienti da sorgenti distinte si riferiscano alla stessa Entità. Nella prima parte della relazione è stato illustrata la pipeline riguardante il pre-processamento dei dati ottenuti dal dataset di AirBnB, in particolare le due fasi riguardano il labeling e lo splitting. Successivamente sono state introdotte le due reti neurali utilizzate con riferimenti alle tecniche utilizzate nella fase di sperimentazione. Nelle ultime sezioni sono stati illustrati i risultati ottenuti utilizzando le metriche quali: Precision, Recall e Fscore. I modelli ottenuti dai 2 addestramenti hanno fornito notevoli risultati ad eccezione di Londra. A seguito di ciò, si potrebbe pensare che per città di grandi dimensioni (quindi con un dataset molto più grande) il sistema non si comporta in modo atteso, comportamento causato da un possibile overfitting durante l'addestramento. Una possibile implementazione futura potrebbe essere quindi quella di addestrare i modelli su dati di training presi da più città (per esempio 40% Roma 30% Londra 30% Amsterdam). Inoltre è necessario evidenziare i valori di Recall ottenuti, infatti abbiamo cercato di mantenere il focus circa la sensibilità del modello cercando di massimizzare i valori di Recall.

References

Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq Joty, Mourad Ouzzani, and Nan Tang. Distributed representations of tuples for entity resolution. *Proc. VLDB Endow.*, 11(11):1454–1467, July 2018. ISSN 2150-8097. doi: DeepEr. URL `https://doi.org/10.14778/3236187.3236198`.

Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. Deep learning for entity matching: A design space exploration. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 19–34, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450347037. doi: 10.1145/3183713.3196926. URL `https://doi.org/10.1145/3183713.3196926`.