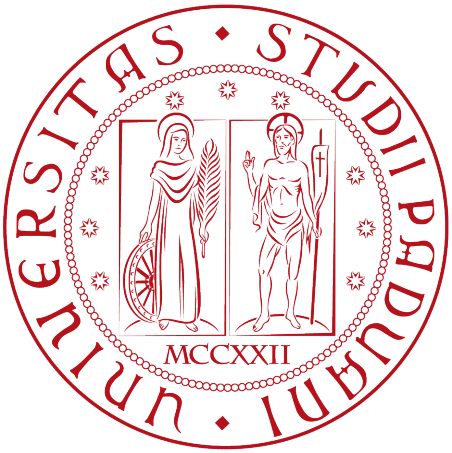
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Confronto di Kernel Methods e Large Language Models nella Sentiment Analysis.

Tesi di laurea

Relatore

Prof. Giovanni Da San Martino

Laureando

Carlo Rosso

Matricola 2034293

Anno Accademico 2023-2024

"I have never let schooling interfere with my education."

— Mark Twain

Dedicato a voi cari lettori, che avete intenzione di leggere questo mio lavoro.

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di tirocinio, della durata di circa trecento ore, dal laureando Carlo Rosso presso l'Università degli Studi di Padova. Gli obiettivi da raggiungere sono stati molteplici.
In primo luogo era richiesto di studiare il problema della *sentiment analysis* e della relativa letteratura scientifica. In secondo luogo era richiesta l'implementazione, sperimentare e confrontare almeno tre modelli allo stato dell'arte sul corpus *Sentiment Penn Treebank*. In terzo luogo era richiesto di documentare i modelli implementati e di caricarli in una *repository* git per garantire la riproducibilità dei risultati. Più nel particolare, durante il periodo di stage è stato indagato il ruolo dell'analisi sintattica e delle sue proprietà di composizionalità nella comprensione dei fenomeni semantici, con un focus specifico sulla *sentiment analysis*.

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine al Prof. Giovanni Da San Martino, relatore della mia tesi, per l'aiuto e il sostegno fornitomi durante la stesura del lavoro. Grazie per avermi guidato e per avermi aiutato ad approfondire argomenti incredibilmente interessanti e stimolanti.

Desidero ringraziare con affetto i miei genitori e tutte le mie famiglie per il sostegno, il grande aiuto e per essermi sempre stati vicini nel cuore ed affettivamente. Grazie per avermi sempre incoraggiato e per avermi spronato nelle scelte fatte e che sto prendendo.

Ho desiderio di ringraziare poi i miei amici. Grazie per aver studiato insieme a me, per avermi supportato e per essermi stati accanto in ogni momento. Soprattutto, grazie per aver stimolato la mia curiosità e per avermi mostrato i vostri punti di vista e le vostre prospettive su tanti argomenti e sul modo di affrontare la vita.

Padova, Settembre 2024 Carlo Rosso

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Il presente lavoro riguarda lo stage svolto presso l'Università degli Studi di Padova, della durata di circa trecentoventi ore, dal 2 luglio al 30 agosto, presso il Dipartimento di Matematica. L'obiettivo del tirocinio è stato lo studio e la comparazione di modelli di classificazione del sentimento utilizzando il Stanford Sentiment Treebank.
Nella prossima sezione verrà descritto il problema e verrà fornita una panoramica generale su cosa significhi classificare il sentimento. Successivamente, verrà introdotto il dataset utilizzato per il confronto tra i diversi modelli. Seguirà una descrizione dettagliata degli obiettivi del tirocinio. Infine, l'ultima sezione illustrerà l'organizzazione del resto del documento.

1.1 Il problema

La classificazione del sentimento è una forma di classificazione del testo in cui un pezzo di testo deve essere classificato in una delle classi di sentimento predefinite. È un problema di apprendimento automatico supervisionato. Nella classificazione binaria del sentimento, le possibili classi sono positivo e negativo. Nella classificazione del sentimento *fine-grained*, ci sono cinque classi (molto negativo, negativo, neutro, positivo e molto positivo).

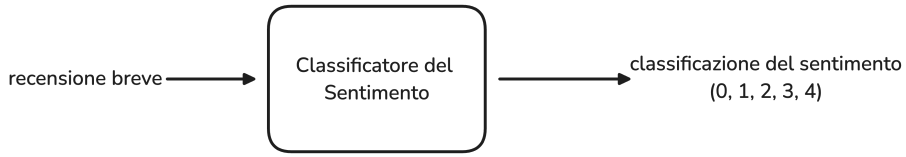


Figura 1.1: *Sentiment analysis*: scatola nera di un classificatore di sentimenti che mostra i sui input e output.

1.2 Dataset

Abbiamo confrontato diversi modelli di classificazione rispetto allo Stanford Sentiment Treebank (Stanford Sentiment Treebank (SST-5))[19].

1.2.1 Origine

Il corpus è stato costruito da stralci di recensioni di film raccolti dal sito `rottentomatoes.com` inizialmente pubblicati da Pang e Lee nel 2005 e consiste in 11.855 periodi. Ciascun periodo è stato analizzato con il parser di Stanford che ha prodotto degli alberi sintattici. Il dataset include un totale di 215.154 frasi uniche derivate da tali alberi sintattici, ciascuna annotata da tre giudici umani.

1.2.2 Struttura

Ciascun esempio è composto da un albero che ha una radice, dei nodi interni e delle foglie che sono etichettati con una di 25 categorie di sentimento: da 0 a 24 (inclusi), dove 0 rappresenta il sentimento più negativo e 24 il più positivo. I giudici hanno commentato che la maggior parte delle frasi potrebbero essere considerate neutre; inoltre, un sentimento più forte viene espresso in frasi che hanno un numero di parole maggiore. Per questo motivo, nello stesso paper in cui hanno presentato il dataset, i ricercatori hanno raggruppato le etichette in cinque classi: molto negativo, negativo, neutro, positivo e molto positivo. Il *treebank* raccoglie frasi in lingua inglese.

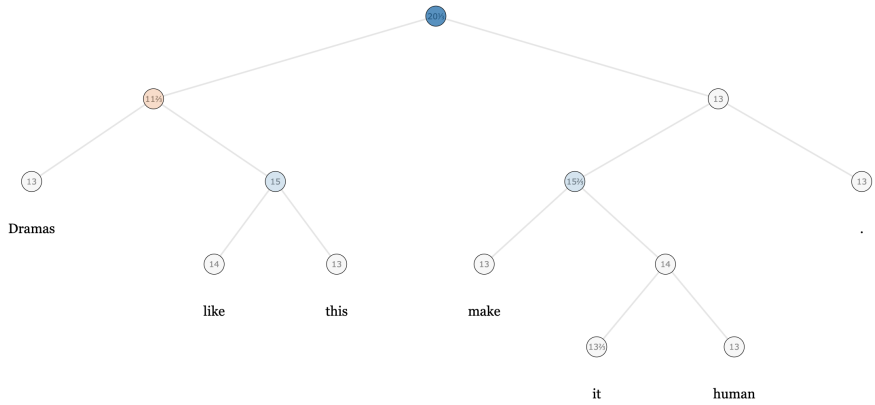


Figura 1.2: Esempio di albero sintattico annotato con le etichette di sentimento.

Il dataset è stato diviso in tre parti: *training set*, *validation set* e *test set*. Il *training set* è utilizzato per allenare il modello, il *validation set* per regolare i parametri del modello e il *test set* per valutare le prestazioni del modello. Come si vede dalla Figura 1.3, il dataset è piuttosto bilanciato e contiene un numero simile di esempi per ciascuna classe. Per questo motivo, abbiamo valutato sull'accuratezza. Si noti che in letteratura si utilizza la medesima metrica per valutare i modelli di classificazione del sentimento su questo dataset[15].

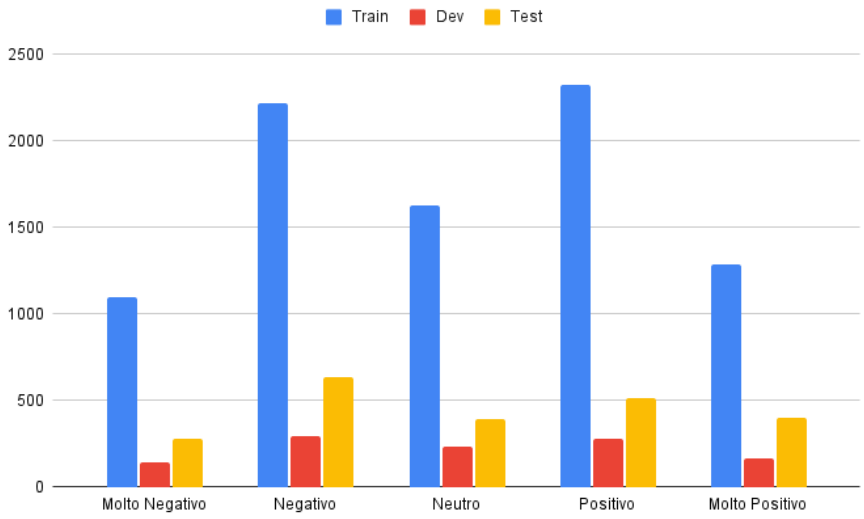


Figura 1.3: Distribuzione delle etichette di sentimento nel dataset.

1.3 Obiettivi

Il tirocinio si è focalizzato sullo studio dello stato dell'arte nella *sentiment analysis* utilizzando il dataset SST-5. Gli obiettivi principali sono suddivisi in quattro punti chiave:

1. **Studio del problema**: comprendere che cosa significhi *sentiment analysis*. Questo aspetto è stato già affrontato precedentemente. Non basta comprendere il problema, è necessario anche identificare e studiare alcuni algoritmi in grado di risolverlo;

2. **Sperimentazione e confronto**: sperimentare e confrontare i modelli individuati. Nello specifico, abbiamo esaminato tre modelli: uno basato sui metodi kernel, uno sulle reti neurali ricorrenti e infine un modello basato sul Trasformer. Ciascun modello verrà approfondito nelle sezioni successive;

3. **Documentazione e riproducibilità**: documentare i risultati ottenuti e organizzare i modelli in una repository GitHub per garantire la riproducibilità dei risultati, fornendo così una base verificabile per le conclusioni tratte. In questo lavoro, ci concentreremo particolarmente sulle differenze emerse tra i vari modelli. Nella repository Github[1] è stato pubblicato il codice sorgente per garantire la riproducibilità dei risultati;

4. **Sviluppo di un modello personalizzato**: sviluppare un modello personalizzato per migliorare lo stato dell'arte. Si tratta di un obiettivo facoltativo non raggiunto.

1.4 Organizzazione del documento

Il secondo capitolo introduce la terminologia e le notazioni utilizzate nel testo, definendo i concetti di base per comprendere la trattazione;

Il terzo capitolo approfondisce il modello basato sui metodi kernel. In particolare per questo e per i prossimi due capitoli, abbiamo descritto il modello e ne abbiamo discusso l'implementazione e i risultati ottenuti;

Il quarto capitolo approfondisce il modello basato sulle reti neurali ricorrenti;

[1] https://github.com/danesinoo/bachelor-s_notes

Il quinto capitolo approfondisce il modello basato sul Trasformer. In particolare abbiamo descritto la struttura del Trasformer vanilla;

Il sesto capitolo trae le conclusioni e discute i risultati ottenuti.

Riguardo la stesura del testo, relativamente al documento sono state adottate le seguenti convenzioni tipografiche:

- gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel glossario, situato alla fine del presente documento;
- per la prima occorrenza dei termini riportati nel glossario viene utilizzata la seguente nomenclatura: *parola*[g];
- i termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*.

Capitolo 2

Notazione

In questo capitolo spieghiamo le notazioni utilizzate nel testo, quindi definiamo i concetti di base per comprendere la trattazione.

2.1 Apprendimento automatico

L'apprendimento automatico comprende tutti gli algoritmi che imparano con l'esperienza, ovvero che migliorano le proprie prestazioni con l'esperienza. In questo modo notiamo che abbiamo bisogno di definire alcuni concetti di base:

- **Pattern**: un pattern è un esempio, l'unità di base dell'esperienza. Nel nostro caso, un pattern è un albero sintattico annotato con le etichette di sentimento;
- **Misura di errore**: una misura di errore è una funzione che misura la distanza tra la funzione che il modello apprende e la funzione obiettivo: la funzione che vogliamo che l'algoritmo apprenda;
- **Accuratezza**: l'accuratezza è il rapporto tra il numero di pattern correttamente classificati e il numero totale di pattern. Noi useremo l'accuratezza per valutare la bontà dei modelli di classificazione del sentimento;
- **Iperparametri**: gli iperparametri sono dei parametri che regolano il comportamento del modello. Gli iperparametri sono fissati prima dell'allenamento e non sono modificati durante l'allenamento;

- **Parametri**: i parametri o pesi sono i valori che il modello modifica durante l'allenamento per modificare il proprio comportamento.

Gli algoritmi di apprendimento automatico che approfondiamo in questo documento apprendono da un dataset, chiamato *training set*, e sono valutati su un dataset disgiunto chiamato *test set*. I pattern all'interno dei due dataset hanno la medesima struttura: contengono un albero sintattico annotato con le etichette di sentimento e il valore obiettivo, che vogliamo che sia predetto. Infatti gli algoritmi che trattiamo sono del tipo supervisionato, ovvero apprendono da esempi già etichettati.
Questi algoritmi di apprendimento automatico sono divisi in due fasi: la fase di apprendimento e la fase di valutazione.
Nella fase di apprendimento il modello riceve in input un pattern alla volta dal *training set* e assegna una categoria all'esempio. Questo risultato viene salvato. Dopo che è stato predetto qualche pattern, viene misurato l'errore sulla classificazione e in base ad esso sono aggiornati i pesi dell'algoritmo. L'aggiornamento dei pesi ha lo scopo di minimizzare l'errore commesso dal modello.
Nella fase di valutazione i parametri sono bloccati, quindi l'algoritmo predice la classe di ciascun pattern. Infine si calcola il rapporto tra il numero di volte in cui l'algoritmo ha azzeccato la classe e il numero di tutti gli esempi, ovvero l'accuratezza. L'accuratezza è utile per confrontare modelli diversi, per comprendere la rilevanza degli iperparametri, ovvero per capire il grado di influenza di un iperparametro sulla bontà del modello; e per individuare il valore ottimale degli iperparametri, ovvero quello che permette di raggiungere la migliore accuratezza.
Il *dev set* ha lo scopo di aiutare ad individuare gli iperparametri ottimali, si noti tuttavia che non tutti gli algoritmi qui proposti utilizzano il *dev set*.

2.2 Alberi

Questa sezione riporta in forma discorsiva le definizioni fondamentali sulle strutture di dati. Un grafo è una coppia $G = (V, E)$, dove V è un insieme di elementi chiamati nodi; mentre E è un insieme di coppie non ordinate di nodi; gli elementi di E sono chiamati archi.

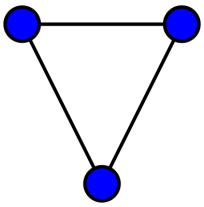


Figura 2.1: Esempio di grafo.

Un percorso $p(v_1, v_n) = (v_1, v_2, \ldots, v_n)$ è una sequenza di nodi tale che $(v_i, v_{i+1}) \in E, 1 \leq i \leq n-1$. Due nodi v_i e v_j sono connessi se esiste un percorso tra v_i e v_j. Un grafo si dice connesso se ogni coppia di nodi è connessa. Un grafo si dice ciclico se esiste esiste almeno un percorso che ritorna al nodo di partenza.

Un albero è un grafo connesso e aciclico. In particolare, noi tratteremo gli alberi con radice, ovvero alberi in cui è presente un nodo speciale chiamato radice. La radice introduce una gerarchia tra i nodi dell'albero. Si noti che la radice è l'unico nodo che non ha archi entranti, mentre le foglie sono i nodi che non hanno archi uscenti.

Un albero binario è un albero in cui ogni nodo ha al massimo due figli. Gli alberi del dataset SST-5 hanno una proprietà ancora più forte: ogni nodo che non è una foglia ha esattamente due figli. Un albero si dice etichettato se a ciascun nodo è associata un'etichetta.

Il dataset SST-5 contiene alberi sintattici, nel senso che ogni frase viene rappresentata con un albero e la forma dell'albero è determinata dalla struttura sintattica della frase; in aggiunta, ogni foglia dell'albero contiene una parola della frase e ogni nodo è etichettato con un valore di sentimento (da 0 a 4). Il valore obiettivo è l'etichetta di sentimento della radice dell'albero.

Sottoalbero Un sottoalbero t è un sottoinsieme di nodi di un albero T, con i corrispondenti archi, tale che t è un albero.

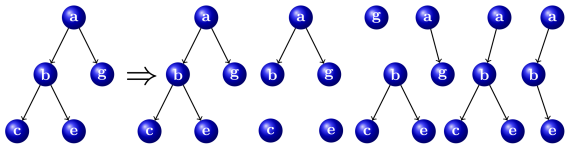


Figura 2.2: Qualche esempio di sottoalbero (a destra), dato un albero (a sinistra).

Sottoalbero proprio Un sottoalbero proprio t è un sottoalbero di un albero T, tale che per ogni nodo v di t, ogni discendente di v in T è in t.

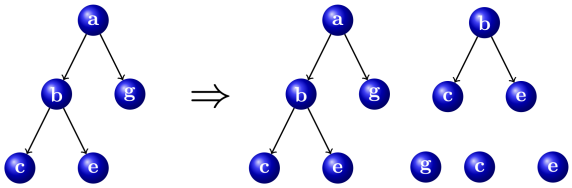


Figura 2.3: Qualche esempio di sottoalbero proprio (a destra), dato un albero (a sinistra).

Subset tree Un *subset tree* è un sottoalbero di un albero T, tale che per ogni nodo v di t, o tutti i figli di v in T sono in t oppure nessuno di essi è in t.

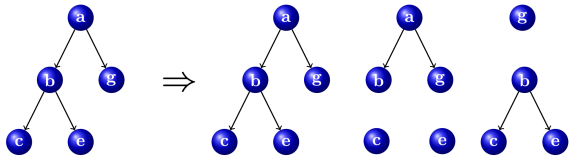


Figura 2.4: Qualche esempio di *subset tree* (a destra), dato un albero (a sinistra).

Capitolo 3

Metodi Kernel

I metodi kernel sono una classe di algoritmi di apprendimento automatico che operano in uno spazio delle caratteristiche non lineare. Questi metodi sono particolarmente adatti per la classificazione di dati strutturati, come ad esempio alberi sintattici (come quelli che si trovano nel corpus SST-5).
In particolare, abbiamo utilizzato la libreria svmlight-tk[12] per allenare e valutare i modelli kernel.
La classe dei metodi kernel comprende tutti quegli algoritmi che non richiedono una rappresentazione esplicita degli esempi ma solo informazioni sulle somiglianze tra di essi. Qualsiasi metodo kernel può essere scomposto in due moduli:

- una funzione kernel specifica per il problema, che calcola le differenze tra i pattern di input;
- un algoritmo di apprendimento generico (nel nostro caso adottiamo *support vector machine*).

Di seguito approfondiamo i due moduli.

3.1 Funzione kernel

Gli input sono confrontati tramite una funzione kernel, che deve essere simmetrica, positiva e semidefinita, in modo che possa essere applicata direttamente nello spazio originale. La funzione kernel è una misura di similitudine tra due pattern di input; per esempio, potrebbe assumere un valore compreso tra 0 e 1, dove 1 indica che i due alberi sono identici e 0 che i

due alberi sono completamente diversi, vedi Figura 3.1.

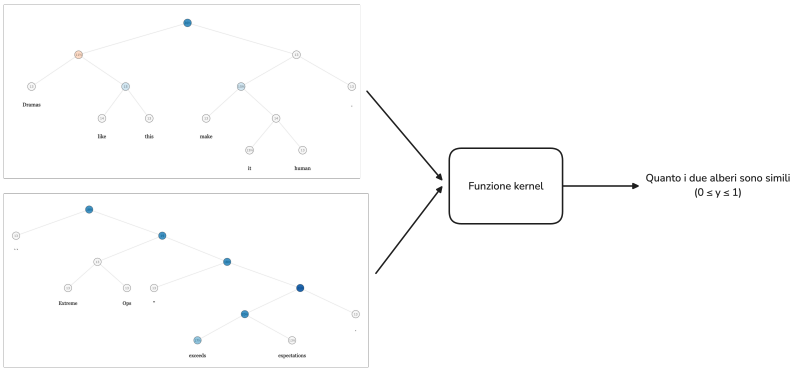


Figura 3.1: Metodo kernel

In breve, la rappresentazione di nuovo esempio viene calcolata tramite una somma ponderata di valutazioni kernel tra l'esempio e un sottoinsieme dei pattern di allenamento già somministrati. In questo modo, la tecnica è in grado di gestire direttamente dati strutturati.
Più avanti valutiamo alcuni metodi kernel, per questo è importante saper valutare la qualità di una funzione kernel, che individuiamo attraverso le seguenti proprietà[9]:

- Completezza: il kernel dovrebbe essere in grado di catturare tutte le informazioni necessarie per la classificazione;
- Correttezza: il kernel dovrebbe essere in grado di catturare solo le informazioni necessarie per la classificazione;
- Adeguatezza: il kernel dovrebbe essere in grado di generalizzare bene anche su nuovi dati. In particolare, pattern in input vicini dovrebbero avere valutazioni kernel simili.

Queste proprietà sono un indice della qualità del kernel, tuttavia è importante notare che bisogna scegliere un kernel che sia adatto al problema che si vuole risolvere; in questo senso è importante considerare anche la complessità computazionale del kernel: un kernel potrebbe richiedere una quantità di risorse computazionali che non abbiamo a disposizione, anche se è molto buono. Questo discorso si applica anche agli altri algoritmi di classificazione.

3.2 *Support Vector Machine*

Le Support Vector Machine (SVM) si basano sul principio della Minimizzazione del Rischio Strutturale[2]. L'SVM è un classificatore binario che proietta gli esempi nello spazio delle caratteristiche e poi cerca un iperpiano che separi gli esempi positivi da quelli negativi. In particolare, viene scelto l'iperpiano che massimizza il margine tra le due classi, vedi Figura 3.2.

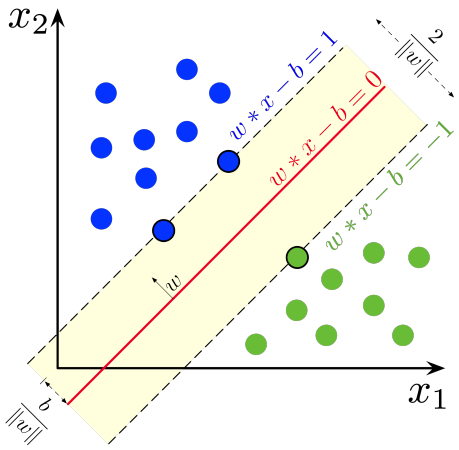


Figura 3.2: Support Vector Machine

3.3 *Softmax*

Come abbiamo già accennato, SST-5 è un problema di classificazione multiclasse, quindi adottiamo la tecnica *one-vs-all* per adattare l'SVM al problema multiclasse. In pratica, si addestrano 5 classificatori binari, uno per ciascuna classe. Per classificare un nuovo esempio, ogni classificatore restituisce un valore tra -1 e 1 che indica la probabilità che l'esempio appartenga alla classe corrispondente oppure che non appartenga ad essa. Il classificatore che restituisce il valore più alto è quello che determina la classe dell'esempio. La funzione che viene applicata, quindi si chiama *softmax*.

3.4 Esempi di funzioni kernel

Fino ad ora abbiamo spiegato in generale come funziona il metodo kernel, ora approfondiamo tre funzioni kernel che abbiamo confrontato sul dataset SST-5. In particolare, vediamo il *subtree kernel*, il *subset tree kernel* e il *partial tree kernel*. I modelli sono ordinati in modo tale per cui si arricchisce sempre di più lo spazio delle caratteristiche. Si noti che tendenzialmente uno spazio delle caratteristiche più descrittivo richiede più risorse computazionali e non è detto che le prestazioni del modello migliorino.

3.4.1 *Subtree kernel*

Il *subtree kernel* restituisce la somma pesata del numero di sottoalberi propri che due alberi hanno in comune. Il kernel è definito come segue:

$$K(T_1, T_2) = \sum_{s \in \mathcal{A}^*} h_s(T_1) h_s(T_2) w_s \tag{3.1}$$

dove $\mathcal{A}^*$ è l'insieme di tutti i sottoalberi propri di T_1 e T_2, $h_s(T)$ è una funzione che restituisce il numero di occorrenze di s in T e w_s è il peso associato al sottoalbero s.

3.4.2 *Subset tree kernel*

Il *subset tree kernel* è una generalizzazione del *subtree kernel*. In particolare, il *subset tree kernel* definisce una misura di similarità tra due alberi che è proporzionale al numero di *subset tree* che i due alberi hanno in comune. Si può dimostrare che la complessità computazionale del *subset tree kernel* è uguale a quella del *subtree kernel* ovvero di $O(|N_{T_1}||N_{T_2}|)$, dove N_{T_1} e N_{T_2} sono il numero di *subset tree* dell'albero T_1 e T_2 rispettivamente.

3.4.3 *Partial tree kernel*

Il *partial tree kernel* è una generalizzazione del *subset tree kernel*. In particolare, il *partial tree kernel* conta il numero di sottoalberi in comune nei due alberi T_1 e T_2. Il *partial tree kernel* ha complessità computazionale $O(\rho^3|N_{T_1}||N_{T_2}|)$, dove ρ è il massimo numero di figli di un nodo dell'albero.

3.5 Implementazione

Come anticipato in precedenza, abbiamo utilizzato la libreria svmlight-tk[12] per allenare e valutare i modelli kernel. La prima parte di questa sezione è quindi dedicata alla descrizione della libreria. La libreria mette a disposizione due programmi principali: `svm_learn` e `svm_classify`, che approfondiamo di seguito.

`svm_learn` Questo programma permette di allenare un modello SVM con una funzione kernel specificata dall'utente. È necessario fornire un file nel formato svmlight, contenente i dati di allenamento organizzati per righe. Ogni riga deve contenere il valore target, che si desidera predire, e una foresta di alberi, ovvero un insieme di alberi. Una descrizione dettagliata del formato è disponibile nella risorsa citata in precedenza. È possibile specificare i parametri del modello tramite diverse *flag*:

- `-W`: specifica se applicare la funzione kernel a tutte le coppie di alberi contenute nella foresta in input e sommare i risultati (valore A); oppure se applicare la funzione kernel in sequenza e sommare i risultati. Ovvero, per ogni foresta F e F', con $|F| = |F'| = n$, si considerano le coppie (f_i, f'_i) con $i = 1, \ldots, n$ e si calcola $K(f_1, f'_1) + \ldots + K(f_n, f'_n)$ (valore S);
- `-F`: specifica la funzione kernel da utilizzare tra:
  - Una variante più ottimizzata del *Partial Tree Kernel* (valore -1);
  - Il *Subtree kernel* (valore 0);
  - Il *Subset Tree Kernel* (valore 1);
  - Il *Subset Tree-Bow Kernel* (valore 2);
- `-M`: fattore di decadimento μ del *Partial Tree Kernel*, con valore predefinito di 0.4;
- `-L`: fattore di decadimento delle funzioni kernel, con valore predefinito di 0.4;
- `-N`: opzione di normalizzazione; il valore 0 indica che non si effettua la normalizzazione, mentre il valore 1 abilita la normalizzazione;
- `-t`: specifica il tipo di funzione kernel da utilizzare, tra cui: combinazione di insiemi di alberi secondo i parametri specificati e *re-ranking* basato sugli alberi, per cui sono necessari due alberi in ciascun input;

Per esempio il comando per allenare un modello potrebbe essere il seguente:

```
./svm_learn -t 3 -F 3 -W S -M 0.4 -L 0.4 -N 1 path/to/dataset.txt
   path/to/output
```

`svm_classify` Questo programma permette di classificare un insieme di alberi utilizzando un modello allenato con `svm_learn` e avendo a disposizione un file nel formato svmlight contenente i dati di test.

Infine, abbiamo implementato uno script in Python che riceve in input il percorso fino ad una cartella che contiene i file di allenamento e di test e il la *flag* `-kernel` che specifica la funzione kernel da utilizzare. Il programma contiene una lista di iperparametri da testare e per ciascuna funzione kernel.
Come abbiamo già spiegato una *support vector machine* è un classificatore binario, mentre il programma sviluppato produce e valuta dei classificatori multiclasse. Dunque la tecnica *one-vs-all* è stata implementata in python e il percorso dei dataset deve avere una struttura ben definita. Infatti, all'interno della cartella ci devono essere almeno i seguenti dataset:

- `multiclass_train.sentiment_<numero_classe>.txt`;
- `multiclass_subtree_test.sentiment_<numero_classe>.txt`;
- `multiclass_test.sentiment_<numero_classe>.txt`.

Ovviamente, uno per ciascuna classe, dove il numero della classe è un numero intero tra 0 e 4.
Per esempio il comando per allenare un modello potrebbe essere il seguente:

```
python3 kernel_method.py -kernel 3 path/to/dataset
```

In questo modo il programma crea una cartella per ciascun modello allenato, per ciascuna combinazione di iperparametri, infine crea un file `results.json` che contiene i risultati ottenuti dalle valutazioni. L'allenamento di ciascun modello di classificazione multiclasse avviene in parallelo e dura più o meno un'ora.

3.6 Dataset

Rispetto all'implementazione bisogna fare una piccola precisazione: le prime righe del programma implementato in python sono dedicate alla definizione delle costanti, tra cui il nome dei file dei dataset di input, come per esempio
`multiclass_train.sentiment_<numero_classe>.txt`. Cambiando tali definizioni è possibile allenare e valutare i modelli su altri dataset. In particolare, a partire dal dataset SST-5 (covertito in formato svmlight) abbiamo costruito altri dataset, per comprendere meglio quali sono le informazioni più rilevanti per la classificazione.
In particolare, abbiamo costruito un dataset `sintax` che sostiuisce le etichette di sentimento con le etichette sintattiche. Questo dataset ci permette di valutare i modelli su un dataset che contiene solo informazioni grammaticali, in questo modo possiamo capire quanto le informazioni sintattiche sono rilevanti per la classificazione.
Poi abbiamo costruito un dataset `merged`, per cui ogni pattern è formato sia da un albero di sentimento che da un albero sintattico.
Infine abbiamo costruito il dataset `subtree` per ciascuno dei dataset così costruito, che contiene tutti i sottoalberi propri di un albero, a meno delle foglie. Questo dataset ci permette di valutare i modelli su ciascun nodo dell'albero, una valutazione più fine che viene svolta anche all'interno dell'articolo di riferimento[19]. Per comparare in modo più preciso i modelli abbiamo quindi costruito questo dataset.
Non è stato possibile costruire i `subtree` che contengono sia le etichette sintattiche che le etichette di sentimento, perché gli alberi sintattici e quelli di sentimento sono lievemente diversi.

3.7 Sperimentazione

Per prima cosa abbiamo confrontato uno stesso modello sui diversi dataset, per capire quali informazioni sono più rilevanti per la classificazione. In particolare, abbiamo utilizzato il *Partial Tree Kernel* per comparare i risultati.

Modello	Accuracy	Precision	Recall	F1
Subtree sentiment all	**0.95**	**0.92**	**0.91**	**0.92**
Subtree sentiment radice	0.51	0.50	0.48	0.49
Merged tutti	0.44	0.48	0.59	0.44
Merged radice	0.53	0.52	0.49	0.49
Sentiment tutti	0.40	0.45	0.56	0.40
Sentiment radice	**0.54**	**0.54**	**0.50**	**0.50**
Syntax tutti	0.18	0.04	0.20	0.06
Syntax radice	0.39	0.38	0.34	0.34

Tabella 3.1: Comparazione dei dataset.

Dopo i risultati ottenuti, abbiamo capito che il dataset etichettato sulla sintassi è di gran lunga il meno rilevante. Invece, permane ancora qualche dubbio rispetto alla differenza tra il dataset sentiment e il dataset merged. Per questo motivo abbiamo confrontato i diversi modelli su entrambi i dataset.

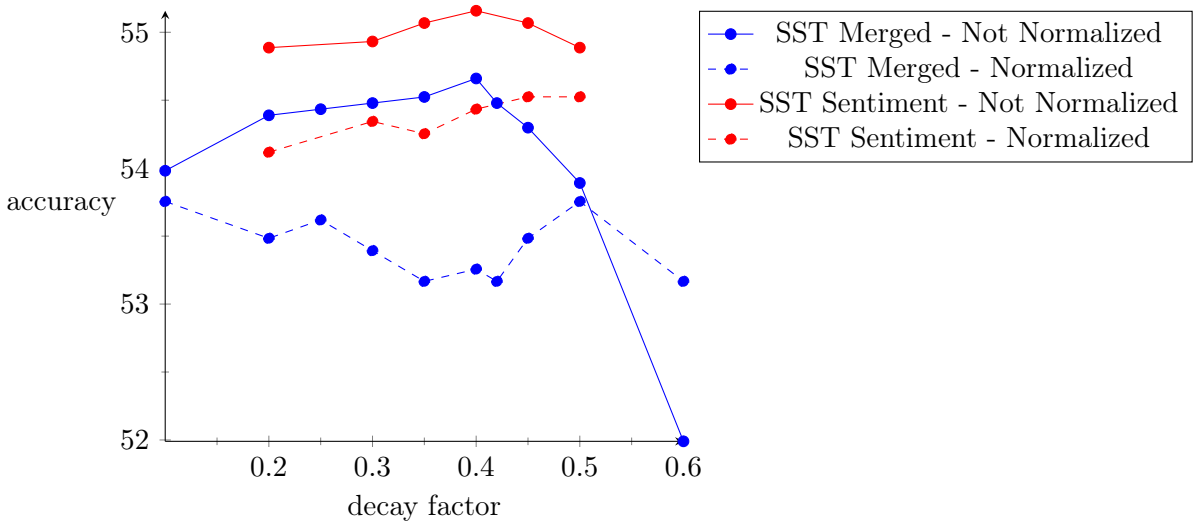


Figura 3.3: Confronto degli iperparametri del subset tree kernel.

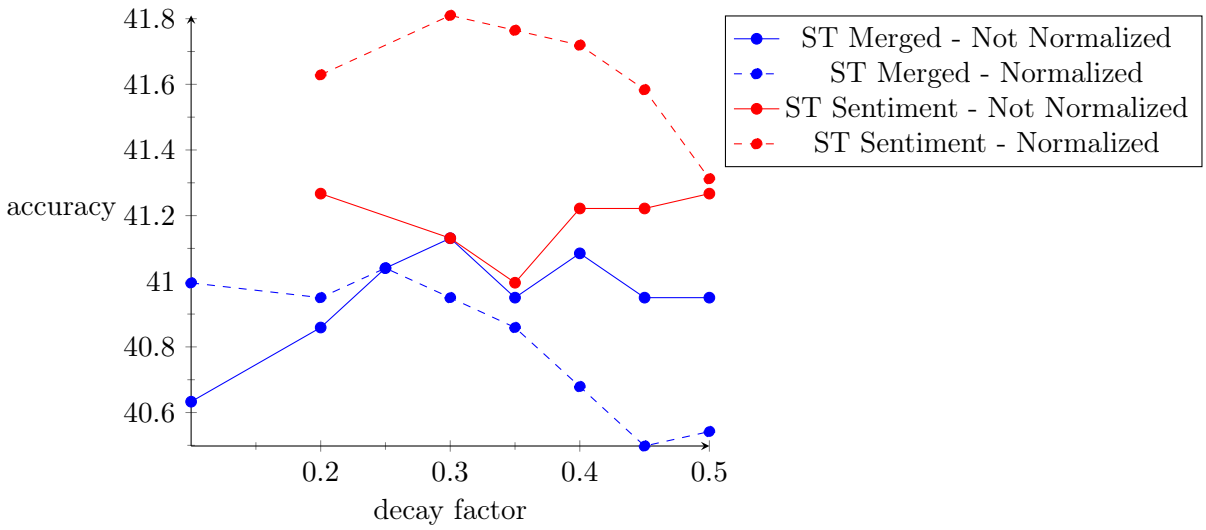


Figura 3.4: Confronto degli iperparametri del subtree kernel.

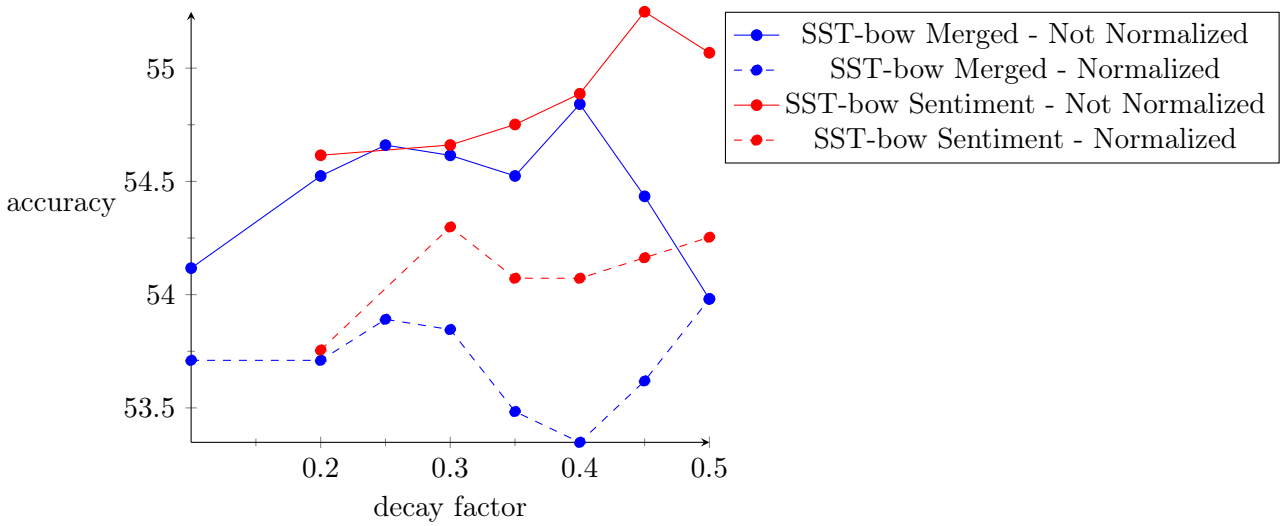


Figura 3.5: Confronto degli iperparametri del subset tree-bow kernel.

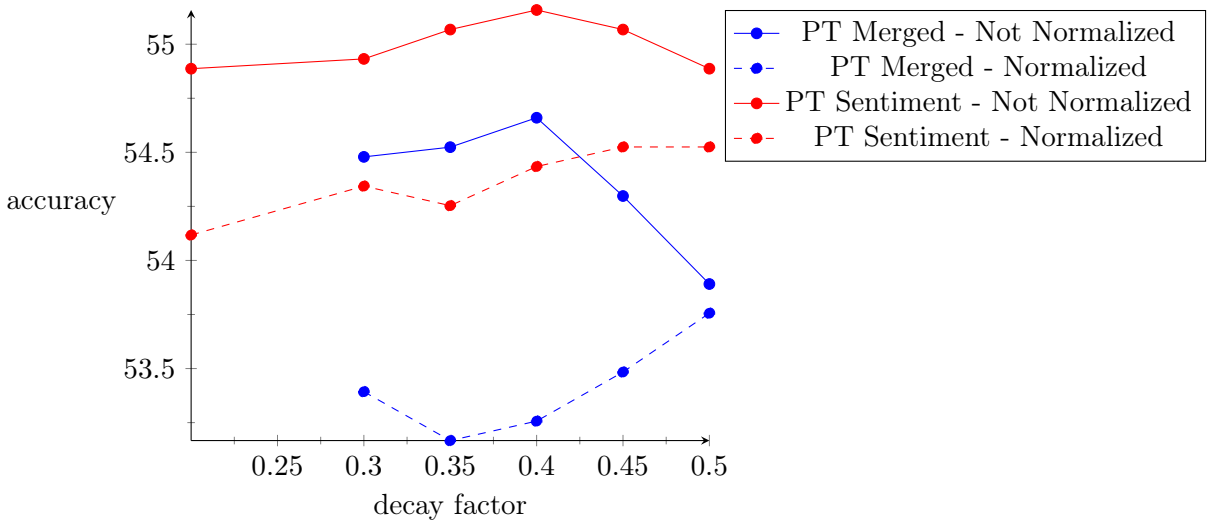


Figura 3.6: Confronto degli iperparametri del partial tree kernel.

3.7.1 Riassumendo

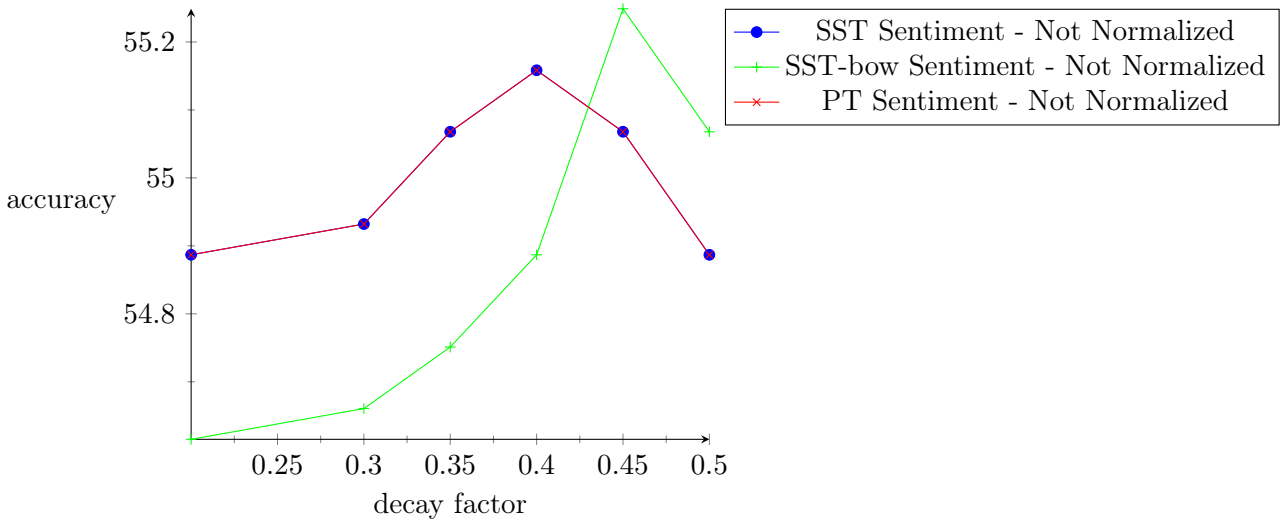


Figura 3.7: Confronto dei metodi kernel sugli iperparametri migliori. Si noti che SST e PT ottengono valutazioni identiche.

I risultati mostrano che l'iperparametro 'W' della libreria svmlight-tk, non è rilevante, infatti qualunque valore assuma non influisce sulle performance del modello. La normalizzazione degli alberi tendenzialmente peggiora le performance dei modelli e ne allunga il tempo di allenamento. Infine, il fattore di decadimento migliore risulta essere di 0.4 per i modelli subset tree, partial tree, mentre il subset tree-bow kernel con decay factor di 0.45 ottiene

un'accuratezza migliore.

Infine notiamo che i modelli subset tree, subset tree-bow e partial tree ottengono performance comparabili: più del 55% di accuratezza nella classificazione dei pattern sulla radice dell'albero. Mentre il modello con kernel subtree raggiunge un'accuratezza poco inferiore al 42%.

Capitolo 4

Recurrent Neural Network

Le Recurrent Neural Network (RNN) sono una classe di reti neurali particolarmente adatte per l'elaborazione di dati sequenziali, come testi, serie temporali e sequenze biologiche. La loro capacità di mantenere una memoria delle informazioni passate le rende strumenti potenti per la modellazione di dipendenze temporali e strutture gerarchiche nei dati. In questo capitolo, esploriamo il funzionamento delle RNN, partendo dal processo di rappresentazione vettoriale delle parole, spostandoci sul passo ricorsivo e sulla fase di classificazione.
Le RNN sfruttano la struttura ad albero dei pattern per classificare una frase. In particolare, il modello prende in input un albero sintattico, dove le foglie rappresentano le parole della frase, e restituisce la classe di sentimento. Dunque, è essenziale mappare ciascuna parola in un vettore di dimensione fissa, consentendo l'applicazione del passo ricorsivo. Questa mappatura, nota come *word embedding*, è cruciale per rappresentare in modo efficace le parole all'interno del modello.
Di seguito, descriviamo come ottenere queste rappresentazioni vettoriali, come combinare i vettori tramite il passo ricorsivo e come utilizzare il vettore risultante per classificare una frase in una delle categorie di sentimento.

4.1 Rappresentazione vettoriale delle parole

Ora spieghiamo come possiamo ottenere un vettore per ciascuna parola. In primo luogo, scegliamo la dimensione del vettore, ad esempio $d = 300$, questo valore è un iperparametro del modello. Poi inizializziamo la matrice dei *word embedding* L con valori casuali. $L \in \mathbb{R}^{d \times |V|}$, dove $|V|$ è la dimensione del vocabolario, ovvero il numero di parole distinte nel corpus. L

è un parametro del modello, ovvero è una variabile che il modello apprende durante la fase di allenamento. Ogni parola è rappresentata da un vettore riga di L.

4.2 Passo ricorsivo

Il passo ricorsivo si occupa di combinare due vettori. All'inizio i vettori che sono combinati rappresentano una parola; mano a mano che i vettori sono combinati, rappresentano una sequenza di parole sempre più lunga, fino a che si ottiene un vettore che rappresenta l'intera frase.
In particolare, il passo ricorsivo prende in input due vettori l e r e restituisce un vettore p, tale che $l, r, p \in \mathbb{R}^d$:

$$p = \tanh\left(W \begin{bmatrix} l \\ r \end{bmatrix}\right) \tag{4.1}$$

Anche in questo caso, W è un parametro del modello. $W \in \mathbb{R}^{d \times 2d}$, ovvero è una matrice che mappa il vettore di input, composto da due *word embedding*, in un vettore di dimensione d. Finalmente, viene applicata una funzione di attivazione, in questo caso la tangente iperbolica, per ottenere il vettore p. La funzione di attivazione ha lo scopo di introdurre non linearità nel modello, in modo tale da regolarizzare il risultato, per renderlo più robusto e interpretabile; infatti, la tangente iperbolica è una funzione non lineare che ritorna valori nell'intervallo $[-1, 1]$.

4.3 Classificazione

Una volta che è stato effettuato il passo ricorsivo per ciascun nodo dell'albero, si ottiene il *word embedding* della radice, ovvero il vettore che rappresenta l'intera frase. Il passo di classificazione ha lo scopo di assegnare una classe tra molto positivo, positivo, neutro, negativo e molto negativo alla frase. In particolare, il vettore della radice viene proiettato in uno spazio di dimensione 5, in cui ciascuna componente rappresenta la probabilità che il *word embedding* appartenga a una delle classi. Infine, viene applicata la funzione *softmax*, che abbiamo destritto nella Sezione 3.3, che estrae la classe di appartenenza della frase con probabilità maggiore, ovvero la predizione del modello. Di seguito riportiamo la formula:

$$y^r = \text{softmax}(W_s r) \tag{4.2}$$

dove r rappresenta il vettore della radice, $W_s \in \mathbb{R}^{5\times d}$ è un parametro del modello e $y^r \in \{0, 1, 2, 3, 4\}$ è la classe predetta.

4.4 Allenamento

L'allenamento ha l'obiettivo di cambiare i parametri del modello in modo che l'algoritmo predica la classe corretta con la massima probabilità. Dunque vogliamo rendere quanto più simile possibile la distribuzione predetta alla distribuzione obiettivo, per questo motivo minimizziamo l'errore di entropia incrociata tra le due distribuzioni.

4.4.1 Calcolo dell'errore

Assumiamo che in ogni nodo, il vettore di distribuzione obiettivo sia rappresentato con una codifica binaria (0,1). Se ci sono C classi, il vettore avrà lunghezza C con un 1 nella posizione corrispondente alla classe corretta e 0 nelle altre posizioni. Il nostro obiettivo è ridurre al minimo l'errore di entropia incrociata tra la distribuzione prevista y_i (che è un vettore di lunghezza C con valori tra 0 e 1) al nodo i e la distribuzione obiettivo t_i (che è un vettore di lunghezza C con valori 0 o 1) in quel nodo. Ridurre l'errore di entropia incrociata equivale a minimizzare la divergenza di Kullback-Leibler (KL) tra le due distribuzioni (a meno di una costante).
Dunque la funzione di errore è definita come segue:

$$E(\Theta) = \sum_i \sum_{j\in C} t_{ij} \log y_{ij} + \lambda ||\Theta||^2 \tag{4.3}$$

dove $\Theta = (L, W, W_s)$ è la tupla dei parametri del modello; ripetiamo che L permette di mappare le parole in vettori, W è la matrice che combina linearmente due *word embedding*, W_s proietta un *word embedding* in un vettore di probabilità. i è l'indice del nodo, mentre $C = \{0, 1, 2, 3, 4\}$ è l'insieme delle classi. t_{ij} è l'elemento j-esimo del vettore obiettivo del nodo i, mentre y_{ij} è l'elemento j-esimo del vettore predetto a partire dall'*embedding* del nodo i. Infine, λ ha lo scopo di regolarizzare il modello, per evitare l'overfitting e per rendere il modello più robusto e generalizzabile.

Si noti che $t_{ij} \log y_{ij}$ ha valore diverso da 0 solo quando $t_{ij} = 1$, cioè per la classe corretta; inoltre, la somma di y_{ij} su j è 1, in quanto y_i è una distribuzione di probabilità. Quindi incrementare y_{ij} per un j vuol dire diminuire y_{ik} per $k \neq j$.

Per minimizzare l'errore, dobbiamo capire in quale modo cambiare i parametri del modello: calcoliamo il gradiente dell'errore rispetto ai parametri del modello, perché il gradiente indica la direzione in cui l'errore diminuisce più velocemente rispetto al parametro considerato. Quindi a ritroso, rispetto al processo di classificazione, calcoliamo in ordine il gradiente dell'errore rispetto a W_s, W e L.

4.4.2 Gradiente di softmax

Sia x_i il *word embedding* sul nodo i. Attraverso l'algoritmo della *backpropagation*, ogni nodo propaga ai nodi figli il proprio errore mediante i pesi di W. Sia $\delta^{i,s} \in \mathbb{R}^d$ l'errore dovuto alla softmax sul nodo i:

$$\delta^{i,s} = \left(W_s^T (y^i - t^i)\right) \otimes \tanh'(x_i) \tag{4.4}$$

dove $f'(x_i)$ è la derivata della tangente iperbolica, e $\otimes$ indica il prodotto tra due vettori elemento per elemento.
Ora vediamo come si calcola il gradiente dell'errore rispetto al parametro W.

4.4.3 Gradiente del passo ricorsivo

Calcoliamo il gradiente di W a partire dalla radice verso le foglie. La derivata completa di W è data dalla somma delle derivate di W in ciascun nodo. Definiamo $\delta^{i,com}$ l'errore totale che arriva al nodo i; si noti che sulla radice questo valore è uguale a $\delta^{r,s}$.
Sia $\delta^{p,com}$ l'errore totale che arriva al nodo p, allora l'errore che p propaga ai figli è:

$$\delta^{p,down} = \left(W^T \delta^{p,com}\right) \otimes \tanh' \left(\begin{bmatrix} l \\ r \end{bmatrix} \right) \tag{4.5}$$

dove l e r sono i vettori dei figli di p. l e r ricevono metà dell'errore totale e ci aggiungono

il proprio errore sulla softmax:

$$\delta^{l,com} = \delta^{l,s} + \delta^{p,down}[0:d] \tag{4.6}$$

$$\delta^{r,com} = \delta^{r,s} + \delta^{p,down}[d+1:2d] \tag{4.7}$$

dove $[i:j]$ indica il vettore che va dall'elemento i-esimo all'elemento j-esimo.
Infine, il gradiente dell'errore rispetto a W è calcolato come segue:

$$\frac{\partial E}{\partial W} = \sum_{v \in V} \delta^{v,com} \begin{bmatrix} l_v \\ r_v \end{bmatrix} \begin{bmatrix} l_v \\ r_v \end{bmatrix}^T \tag{4.8}$$

Dove V è l'insieme dei nodi dell'albero, l_v e r_v sono gli *embedding* dei figli sinistro e destro del nodo v rispettivamente.

Si noti che in questo modo utiliziamo la medesima matrice W per tutti i nodi dell'albero, in quanto il modello assume che la stessa matrice possa combinare in modo efficace i vettori di *word embedding* in ogni nodo dell'albero. Inoltre, il modello sfrutta le etichette di sentimento disponibili in ciascun nodo dell'albero per imparare i pesi di W. Questo approccio rende l'algoritmo molto semplice, efficiente e scalabile.
L'unico dilemma da porsi è se la combinazione lineare di una coppia di vettori sia una rappresentazione sufficientemente espressiva per catturare le parole e le loro relazioni all'interno della frase.

4.5 Implementazione

Viene richiesto di implementare la *Recursive Neural Network* (RNN) descritta nel paper [19] Quindi l'implementazione viene svolta a partire dal codice fornito dagli autori del paper. In particolare seguono i passaggi svolti per l'implementazione:

1. **Clone del repository**: abbiamo clonato il repository fornito dagli autori del paper;
2. **Installazione delle dipendenze**: abbiamo seguito le istruzioni nel README del repository per installare le dipendenze necessarie;
3. **Check del modello fornito**: abbiamo verificato che il modello fornito funzionasse correttamente con il comando

```
java -cp "*" -mx5g edu.stanford.nlp.sentiment.SentimentPipeline
-file examples/sample-maven-project/sample-english.txt
```

Non ha funzionato;

4. **Correzione**: abbiamo sostituito il comando con

   ```
   mvn exec:java -Dexec.mainClass="edu.stanford.nlp.sentiment.SentimentPipeline"
   -Dexec.args="-file examples/sample-maven-project/sample-english.txt"
   ```

   e ha funzionato;

5. **Implementazione del modello**: abbiamo individuato la classe che si occupa dell'allenamento del modello, ovvero `SentimentTraining`;

6. **Costruzione del comando di allenamento**: abbiamo costruito il seguente comando per allenare il modello:

   mvn exec:java -Dexec.mainClass="edu.stanford.nlp.sentiment.SentimentTraining"
   -Dexec.args="-train -model rnn.ser.gz -trainpath train.txt -devpath dev.txt
   -nousetensors -lowercasewordvectors -numhid 100 -randomseed 42"

   In particolare, la classe specificata si occupa di allenare un modello generico. Di seguito spiego gli argomenti passati al comando:

   - `-train`: specifica che si vuole allenare il modello;
   - `-model ../rnn.ser.gz`: specifica il percorso in cui salvare il modello allenato;
   - `-trainpath ../train.txt`: specifica il percorso del file o della directory di training;
   - `-devpath ../dev.txt`: specifica il percorso del file o della directory di validazione;
   - `-nousetensors`: specifica che non si vogliono utilizzare i tensori. In questo modo si passa dall'RNTN descritto nel paper all'RNN, proprio come spiegato all'interno del paper medesimo;
   - `-lowercasewordvectors`: specifica che si vogliono utilizzare i vettori delle parole in minuscolo, si tratta di una scelta personale, a dire il vero non so se sia stata usata anche dagli autori del paper;
   - `-numhid 100`: specifica il numero di hidden units;

- `-randomseed 42`: specifica il seed per la generazione dei numeri casuali, in modo da rendere riproducibile l'esperimento.

7. **Setup dei dataset**: abbiamo scaricato i dataset nel formato opportuno: si trovano alla seguente risorsa[17].

8. **Allenamento del modello**: abbiamo eseguito il comando di allenamento del modello. Il training è durato circa 6 ore;

9. **Test del modello**: abbiamo testato il modello con il comando
- mvn exec:java -Dexec.mainClass="edu.stanford.nlp.sentiment.Evaluate"
-Dexec.args="-model rnn.ser.gz -treebank test.txt" .

4.6 Sperimentazione

Modello	Accuracy	Precision	Recall	F1
RNN [19] tutti	79.0	-	-	-
RNN [19] radice	**43.2**	-	-	-
RNN con 25 unità nascoste tutti	78.5	58.0	58.7	58.0
RNN con 25 unità nascoste radice	39.7	39.1	39.4	38.1
RNN con 50 unità nascoste tutti	79.4	65.2	53.2	56.9
RNN con 50 unità nascoste radice	42.4	44.3	39.3	40.1
RNN con 75 unità nascoste tutti	79.2	63.2	53.2	56.9
RNN con 75 unità nascoste radice	38.8	41.2	38.7	38.4
RNN con 100 unità nascoste tutti	78.2	60.1	56.7	57.2
RNN con 100 unità nascoste radice	39.8	38.6	37.8	35.6

Tabella 4.1: Comparazione di RNN per numero di unità.

I risultati ottenuti confermano le affermazioni dell'articolo[19]. Infatti, il modello RNN dimostra una notevole robustezza anche variando il numero di unità nascoste. Inoltre, l'accuratezza sui singoli nodi si attesta intorno all'80%, in linea con quanto riportato nel paper, che indica un'accuratezza del 79.0%. Nel nostro esperimento, il modello con 50 unità nascoste ha raggiunto un'accuratezza del 79.4%, un valore molto vicino a quello del paper.

Anche l'accuratezza relativa alle etichette della radice è comparabile: l'articolo riporta un'accuratezza del 43.2%, mentre il nostro modello con 50 unità nascoste ha raggiunto il 42.2%. Questi risultati suggeriscono che il modello è stato allenato correttamente e che le prestazioni ottenute sono soddisfacenti. Le differenze riscontrate potrebbero essere dovute all'inizializzazione dei pesi, poiché il seed utilizzato nel paper non è stato fornito, impedendoci di valutare se tali differenze derivino da questo aspetto.

Capitolo 5

Transformer

Il modello chiamato Transformer si è dimostrato estremamente efficace nei problemi di elaborazione del linguaggio naturale, noti come Natural Language Processing (NLP). Questo algoritmo si basa su una rete neurale che utilizza il meccanismo di *self-attention* per elaborare i dati in input e identificare le relazioni tra le parole. Tale meccanismo ha una complessità computazionale quadratica, poiché per ogni parola in input è necessario calcolare l'attenzione rispetto a tutte le altre parole. In letteratura, esistono diverse varianti del Transformer progettate per ridurre questa complessità computazionale, come ad esempio *BP-Partitioning*[23] e *Star-Transformer*[5].

Il meccanismo di attenzione tra le parole, che determina quanto ciascuna parola sia rilevante rispetto alle altre, viene calcolato da zero; di conseguenza, i Transformer tendono a performare molto bene su grandi dataset, ma possono incorrere nel fenomeno dell'overfitting quando applicati a dataset più piccoli[5].

Per ovviare a questo problema, è possibile allenare Transformer di grandi dimensioni su dataset molto estesi, come nel caso di modelli noti quali BERT, RoBERTa o il più recente e famoso GPT. Successivamente, si può eseguire il *fine-tuning* su dataset più piccoli: in questo processo si aggiungono alcuni strati finali al Transformer già addestrato e si allenano solo questi strati su un dataset più piccolo, utilizzando un numero di epoche molto ridotto, ma con un numero elevato di batch. Questo approccio permette agli strati sottostanti di definire la rappresentazione delle parole in input, mentre gli strati finali, più piccoli e semplici, consentono di adattare il modello alle specifiche esigenze del problema. Inoltre, tale tecnica facilita il trasferimento della conoscenza appresa dal transformer alla base,

rendendolo riutilizzabile per una vasta gamma di problemi specifici. Questo approccio è molto interessante perché permette di ridurre i tempi di addestramento e quindi anche i costi computazionali ed economici associati.
Di seguito spieghiamo il funzionamento del Transformer di base, introdotto nel paper [22]. Poi mi limito a spiegare l'implementazione e i risultati ottenuti dai modelli che abbiamo usato per il nostro progetto.

5.1 Architettura

Il modello Transformer ha rivoluzionato il campo delle NLP introducendo un meccanismo basato sull'attenzione per elaborare sequenze di dati, come le parole. Di seguito, descriviamo l'architettura del Transformer seguendo l'ordine di elaborazione dei dati in input.

Tokenizzazione Il primo passo è la tokenizzazione, che utilizza un layer di *embedding* per convertire ciascuna parola in un vettore numerico denso. A questa rappresentazione numerica viene aggiunto un vettore che rappresenta la posizione della parola nella frase, creando così un token per ogni parola in input.

Multi-head self-attention Il meccanismo di *self-attention* consente a ogni token di prestare attenzione a tutti gli altri token nella sequenza, permettendo al modello di catturare le dipendenze indipendentemente dalla loro distanza. Questo avviene calcolando dei punteggi che pesano l'importanza del contributo di ciascun token alla rappresentazione degli altri.
Il *multi-head self-attention*, invece, permette di avere più meccanismi di attenzione in parallelo, ciascuno con un set di pesi diverso. Questo consente al modello di apprendere vari tipi di relazioni tra le parole.
Una possibile interpretazione intuitiva è che una testa di attenzione potrebbe cogliere la relazione tra un nome e i suoi aggettivi, mentre un'altra testa potrebbe rilevare la relazione tra soggetto, verbo e complemento oggetto, e così via. È importante notare che le relazioni tra le parole individuate dal modello non sono predefinite e non è detto che siano sempre semanticamente o sintatticamente significative; sono quelle che il modello apprende durante l'addestramento.
Segue una rappresentazione matematica del meccanismo. Consideriamo una frase composta da n token (cioè parole convertite in vettori numerici); il Transformer calcola iterativamente,

al layer t, il valore di ciascun token di dimensione d, $H_i^t \in \mathbb{R}^d$, dove H^0 rappresenta l'output del layer di tokenizzazione. Il meccanismo chiave del Transformer, noto come *Multi-head self-attention* (MSA), è formulato come segue:

$$\begin{aligned} \mathrm{MSA}(H) &= [\mathrm{head}_1, \ldots, \mathrm{head}_h]\, W^O, \\ \mathrm{head}_i &= \mathrm{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d}}\right) V_i, \\ Q_i = HW_i^Q, \quad K_i &= HW_i^K, \quad V_i = HW_i^V \end{aligned}$$

Dove h è il numero di teste di attenzione, e W_i^Q, W_i^K, W_i^V, W^O sono matrici di pesi apprese durante l'addestramento.

Feed-Forward networks Ogni layer del Transformer è composto da due sottolayer: il primo calcola l'attenzione tra i token e la normalizza, mentre il secondo è un layer di stabilizzazione, che combina l'input del layer corrente con il risultato del meccanismo di attenzione normalizzato e applica un'ulteriore normalizzazione.
Il Transformer calcola quindi i pesi del layer H^{t+1} a partire da H^t nel modo seguente:

$$Z^t = \mathrm{norm}\left(H^t + \mathrm{MSA}(H^t)\right), \tag{5.1}$$

$$H^{t+1} = \mathrm{norm}\left(Z^t + \mathrm{FFN}(Z^t)\right) \tag{5.2}$$

Dove $\mathrm{norm}(\cdot)$ rappresenta un layer di normalizzazione e $\mathrm{FFN}(\cdot)$ è una *feed-forward network* applicata ai vettori posizionali.

Il Transformer è costituito da un numero di layer L applicati in sequenza. Si noti che ogni layer t possiede i propri pesi distinti. Ogni layer è identico agli altri, eccetto per i pesi che vengono appresi durante l'addestramento. Questa struttura permette di creare rappresentazioni sempre più astratte dei dati in input, con ogni layer che influenza la rappresentazione di ciascun token considerando l'intera frase.
Avendo descritto l'architettura del Transformer, prendiamo ad esempio il modello BERT, che è una variante del Transformer preallenata su un corpus molto ampio e vediamo come abbiamo usato questo modello all'interno del nostro progetto.

5.2 BERT

Bidirectional Encoder Representations from Transformers (BERT) è un modello di *embedding* progettato per allenare rappresentazioni bidirezionali delle parole, considerando sia il contesto precedente che quello successivo alla parola in esame[13]. È particolarmente interessante notare che BERT è allenato in modo non supervisionato, il che rende più semplice il suo addestramento rispetto ai modelli che richiedono un corpus etichettato (ricordiamo che i modelli esaminati precedentemente erano supervisionati).
BERT è stato pre-allenato su due obiettivi principali:

- *Masked Word Prediction*: il 15% delle parole in input viene mascherato e il modello impara a prevedere le parole mancanti.
- *Next Sentence Prediction*: il modello riceve due frasi in input e impara a prevedere se la seconda frase segue la prima nel testo originale.

Grazie all'architettura del Transformer, BERT è in grado di processare ciascun token in input in parallelo, riducendo significativamente i tempi di addestramento. BERT può essere utilizzato per il compito di analisi del sentimento aggiungendo semplicemente uno strato di classificazione finale ed eseguendo il *fine-tuning* sul dataset etichettato. Nel repository del progetto è presente un notebook che mostra come abbiamo implementato BERT per il nostro caso d'uso. Di seguito è fornita una breve introduzione al codice utilizzato.

5.3 Implementazione

Per iniziare, importiamo il database delle recensioni, che presenta un formato diverso rispetto a quello utilizzato dai modelli precedenti. BERT, infatti, non può trattare dati strutturati, ma solo testo puro. Il dataset è stato fornito da Hugging Face[4].
Successivamente, utiliziamo la libreria `transformers` per importare il modello BERT pre-addestrato e il corrispondente tokenizzatore. È importante notare che la libreria `transformers`, anch'essa fornita da Hugging Face[3], consente di importare anche altri modelli. Questa libreria fornisce un'interfaccia semplice e intuitiva per caricare un modello pre-addestrato e aggiungere un layer di classificazione. A tal fine, abbiamo utilizzato la classe `AutoModelForSequenceClassification` con il metodo `from_pretrained`, che richiede due parametri in input: il nome del modello e il numero di classi di output (5 nel nostro caso).

In questo progetto abbiamo utilizzato i seguenti modelli: BERT, richiamato con il nome `bert-base-uncased`, RoBERTa, con il nome `roberta-base`, e DistilBERT, con il nome `distilbert-base-uncased`.
Inizialmente, abbiamo allenato il modello in locale; l'allenamento dura circa 1 ora, ma viene parallelizzato dal sistema operativo e viene diviso su 4 core.
Avendo ottenuto questo dato, abbiamo deciso di eseguire l'allenamento dei modelli su Google Colab, che offre gratuitamente l'utilizzo di una GPU, che riduce drasticamente il tempo di allenamento: circa 6 minuti per ogni modello.

5.4 Sperimentazione

Di seguito riportiamo i risultati ottenuti dai modelli BERT, RoBERTa e DistilBERT, arrotondando l'accuratezza alla prima cifra decimale. Ciascun modello è stato allenato per 3 epoche, perché abbiamo notato che i modelli cominciano a soffrire di *overfitting* molto velocemente.

Modello	*Learning Rate*	*Batch Size*	Accuratezza (maggiore è meglio)
BERT[13]	-	-	52.3
BERT	0.0001	16	47.1
BERT	e^{-5}	16	52.3
BERT	e^{-6}	16	50.6
BERT	0.0001	32	48.1
BERT	e^{-5}	32	53.2
BERT	e^{-6}	32	44.6
RoBERTa[21]	-	-	56.4
RoBERTa	0.0001	16	23.0
RoBERTa	e^{-5}	16	56.7
RoBERTa	e^{-6}	16	54.6
RoBERTa	0.0001	32	51.8
RoBERTa	e^{-5}	32	**57.3**
RoBERTa	e^{-6}	32	53.0
DistilBERT	0.0001	16	44.3
DistilBERT	e^{-5}	16	52.0
DistilBERT	e^{-6}	16	47.9
DistilBERT	0.0001	32	48.3
DistilBERT	e^{-5}	32	51.9
DistilBERT	e^{-6}	32	45.5

Tabella 5.1: Risultati ottenuti dai modelli BERT, RoBERTa e DistilBERT.

Notiamo che RoBERTa ha ottenuto i risultati migliori, con un'accuratezza del 57.3%. In realtà, eseguendo più volte il *fine-tuning* con gli stessi iperparametri, il risultato cambia leggermente, tuttavia, in generale, sembra che RoBERTa sia il modello più performante tra quelli testati e che gli iperparametri migliori siano un *learning rate* di e^{-5} e un *batch size* di 32, questo risultato viene confermato anche dagli altri modelli, anche se ottengono risultati inferiori. Infine, notiamo che RoBERTa e BERT hanno la medesima architettura, ma RoBERTa è stato allenato su un corpus più ampio che migliora le prestazioni. Mentre DistilBERT è una versione più leggera di BERT, che riduce la complessità computazionale

e il numero di parametri, per cui ci aspettiamo che abbia un'accuratezza inferiore rispetto agli altri due modelli.

Capitolo 6

Conclusioni

In questa sezione, riassumiamo i risultati ottenuti durante il tirocinio, evidenziando gli obiettivi raggiunti e fornendo un giudizio personale sul lavoro svolto.

6.1 Risultati ottenuti

Nel corso del tirocinio, abbiamo approfondito il concetto di *sentiment analysis*. Successivamente, abbiamo confrontato tre classi di algoritmi allo stato dell'arte:

- **Metodi kernel**: In particolare, abbiamo esaminato e confrontato vari metodi kernel, tra cui il *subtree kernel*, il *subset tree kernel*, il *partial tree kernel* e il *subset tree-bow kernel*;
- **Reti Neurali Ricorrenti**: In questa classe, ci siamo concentrati principalmente sul modello di base delle reti neurali ricorrenti, variando il numero di unità nascoste. Avremmo potuto includere nel confronto anche una generalizzazione del modello RNN, come il Recurrent Neural Tensor Network [19];
- **Modelli Basati su Transformer**: Abbiamo analizzato tre modelli basati su Transformer, ovvero BERT, RoBERTa e DistilBERT, ciascuno con uno strato di classificazione finale. Sarebbe stato interessante estendere il confronto includendo un *interpretation layer*, come proposto in [21]. Altri modelli, come GPT-2 o LLaMA 3, disponibili su HuggingFace [3], avrebbero potuto essere considerati per ulteriori confronti.

Nel complesso, riteniamo che il lavoro svolto sia stato più che soddisfacente, in quanto ha permesso di confrontare le tre classi di modelli, come richiesto. Inoltre, per due delle tre classi, abbiamo sperimentato diverse varianti dei modelli, arricchendo così l'analisi.
Infine, pur non essendo richiesto dagli obiettivi iniziali, abbiamo utilizzato i metodi kernel per analizzare l'influenza della struttura grammaticale nella classificazione del sentimento delle recensioni.

Modello	**Accuratezza**
SST-bow Sentiment	55.2
RNN con 50 unità nascoste	42.4
RoBERTa	**57.3**

Tabella 6.1: Riepilogo dei risultati ottenuti.

Confrontando il modello migliore di ciascuna classe, abbiamo osservato che RoBERTa ha ottenuto un'accuratezza superiore rispetto agli altri due modelli, nonostante utilizzi un dataset con meno informazioni. Questo risultato può essere attribuito al fatto che RoBERTa è un modello pre-allenato su un dataset molto più ampio rispetto a SST-5. Questo risultato evidenzia l'importanza e l'utilità del trasferimento di conoscenza e del riutilizzo di modelli preaddestrati.
Un risultato più inaspettato è emerso dal modello basato sui metodi kernel, che sfrutta intensivamente la struttura ad albero per rendere ciascun pattern linearmente separabile nello spazio delle *feature*. Questo approccio ha mostrato risultati significativi, dimostrando che la struttura grammaticale ha un impatto rilevante nella classificazione del sentimento.

6.2 Sviluppi futuri

Non siamo riusciti a realizzare un modello personalizzato, un risultato che ci dispiace, ma riteniamo comunque che il lavoro svolto sia stato soddisfacente.
In ogni caso, un possibile sviluppo futuro potrebbe essere la creazione di un modello ibrido che utilizzi inizialmente RoBERTa per classificare le recensioni. Parallelamente, un altro modello basato sui metodi kernel potrebbe essere sviluppato per la classificazione. Successivamente, si potrebbe implementare un layer di *voting* sopra questi due modelli per

classificare le recensioni, sfruttando così le potenzialità di entrambi i metodi.
Un altro approccio personalizzato potrebbe coinvolgere un RNN in cui l'embedding è fornito dal tokenizzatore di RoBERTa. A partire da questo modello, si potrebbero confrontare diversi tokenizzatori per determinare quale sia più adatto alla classificazione del sentimento. Un'ulteriore possibilità sarebbe l'integrazione di un layer in ogni modello, allenato per riconoscere quando il modello di base commette errori, utilizzando i metodi kernel. Se il layer rileva un errore, un altro layer, specificamente allenato sui pattern che il modello di base non riesce a classificare correttamente, interverrebbe per correggerlo. Tuttavia, questo approccio potrebbe risultare poco pratico a causa della complessità che comporterebbe.
Infine, un modello molto interessante potrebbe essere costituito da un transformer o da un metodo kernel allenato su SST-2, quindi sullo stesso dataset, ma per la classificazione binaria. Una volta che il modello classifica le recensioni come positive o negative, si potrebbero allenare due ulteriori layer: uno per classificare le recensioni positive e l'altro per quelle negative. Questo approccio sembra promettente, poiché tutti i modelli studiati raggiungono elevate prestazioni nella classificazione binaria e il *bias* introdotto da questa modifica permette di tenere in considerazione l'ordine delle classi.

6.3 Competenze acquisite

Questo tirocinio mi ha permesso di consolidare le mie conoscenze nel campo delle NLP e di approfondire la comprensione di alcuni modelli allo stato dell'arte. Ho imparato a confrontare i modelli e a valutarli utilizzando metriche appropriate, oltre a documentare i risultati e caricare i modelli su una repository GitHub. Ho sviluppato la capacità di documentarmi in modo autonomo, di leggere e comprendere articoli scientifici, e di confrontare i risultati ottenuti con quelli presenti in letteratura. Inoltre, ho imparato a stabilire scadenze e a rispettarle, organizzando il mio tempo per raggiungere gli obiettivi prefissati. Questo tirocinio mi ha anche insegnato a lavorare in autonomia e a risolvere problemi in modo efficace.

6.4 Valutazione personale

Ritengo che questo tirocinio sia stato estremamente formativo e utile per la mia carriera accademica e professionale. Sono convinto che le competenze acquisite saranno di grande valore in futuro, e considero il lavoro svolto come altamente soddisfacente. Grazie a questa esperienza, ho avuto l'opportunità di confrontare tre classi di modelli allo stato dell'arte, il che mi ha richiesto di documentarmi su ciascuno di essi e di condurre un'analisi comparativa approfondita. Credo che queste competenze saranno particolarmente utili durante i miei studi magistrali e nel prosieguo della mia carriera, permettendomi di dimostrare il mio interesse e la mia competenza nel campo della *sentiment analysis* e delle NLP.

Acronimi e abbreviazioni

BERT BERT. 32, 41

NLP NLP. 29, 41

RNN RNN. 21, 41

SST-5 SST-5. 2, 41

SVM SVM. 12, 41

Glossario

BERT è l'acronimo di *Bidirectional Encoder Representations from Transformers*. Si tratta di un modello di *deep learning* per le NLP sviluppato da Google. BERT è stato allenato su un corpus di testo molto vasto e ha ottenuto risultati molto buoni in molte sue applicazioni. Infatti anche Facebook, partendo da questo modello, ha sviluppato RoBERTa, che ha la medesima architettura, ma è allenato su un corpus più vasto. 40

NLP è l'acronimo di *Natural Language Processing*, che si traduce come "Elaborazioen del Linguaggio Naturale". Si riferisce al campo dell'intelligenza artificiale che si occupa dell'interazione tra il computer e il linguaggio usato dagli esseri umani. Alcune applicazioni sono la classificazione di testi (tratta in questa tesi), la traduzione automatica oppure il riassuno di qualche testo, tra le altre. 40

RNN è l'acronimo di *Recurrent Neural Network*. Si tratta di un tipo di rete neurale che è in grado di elaborare sequenze di dati. Approfondiamo questa voce al capitolo 4. 40

SST-5 si riferisce a Stanford Sentiment Treebank ed è un dataset di recensioni di film etichettate con il loro sentimento. Le etichette sono state assegnate da 3 giudici. Il dataset è composto da 11.855 recensioni, divise in 5 classi di sentimenti: molto negativo, negativo, neutro, positivo, molto positivo. 40

SVM è l'acronimo di *Support Vector Machine*. Si tratta di un algoritmo di apprendimento automatico che è in grado di classificare i dati in due classi. Approfondiamo questa voce alla sezione 3.2. 40

Bibliografia

Riferimenti bibliografici

[5] Qipeng Guo et al. «Star-Transformer». In: *CoRR* abs/1902.09113 (2019). arXiv: 1902.09113. URL: http://arxiv.org/abs/1902.09113 (cit. a p. 29).

[6] Franz A. Heinsen. «An Algorithm for Routing Capsules in All Domains». In: *CoRR* abs/1911.00792 (2019). arXiv: 1911.00792. URL: http://arxiv.org/abs/1911.00792.

[9] Giovanni Da San Martino. «Kernel Methods for Tree Structured Data». Tesi di dott. alma, apr. 2009. URL: http://amsdottorato.unibo.it/1400/ (cit. a p. 11).

[10] Alessandro Moschitti. «Kernel-Based Machines for Abstract and Easy Modeling of Automatic Learning». In: *Formal Methods for Eternal Networked Software Systems: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*. A cura di Marco Bernardo e Valérie Issarny. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 458–503. ISBN: 978-3-642-21455-4. DOI: 10.1007/978-3-642-21455-4_14. URL: https://doi.org/10.1007/978-3-642-21455-4_14.

[13] Manish Munikar, Sushil Shakya e Aakash Shrestha. «Fine-grained Sentiment Classification using BERT». In: *CoRR* abs/1910.03474 (2019). arXiv: 1910.03474. URL: http://arxiv.org/abs/1910.03474 (cit. alle pp. 32, 34).

[16] Matthew E. Peters et al. «Deep contextualized word representations». In: *CoRR* abs/1802.05365 (2018). arXiv: 1802.05365. URL: http://arxiv.org/abs/1802.05365.

[18] Richard Socher et al. «Parsing natural scenes and natural language with recursive neural networks». In: *Proceedings of the 28th International Conference on International Conference on Machine Learning*. ICML'11. Bellevue, Washington, USA: Omnipress, 2011, pp. 129–136. ISBN: 9781450306195.

[19] Richard Socher et al. «Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank». In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. A cura di David Yarowsky et al. Seattle, Washington, USA: Association for Computational Linguistics, ott. 2013, pp. 1631–1642. URL: https://aclanthology.org/D13-1170 (cit. alle pp. 2, 16, 25, 27, 36).

[21] Zijun Sun et al. «Self-Explaining Structures Improve NLP Models». In: *CoRR* abs/2012.01786 (2020). arXiv: 2012.01786. URL: https://arxiv.org/abs/2012.01786 (cit. alle pp. 34, 36).

[22] Ashish Vaswani et al. «Attention Is All You Need». In: (2023). URL: https://arxiv.org/abs/1706.03762 (cit. a p. 30).

[23] Zihao Ye et al. «BP-Transformer: Modelling Long-Range Context via Binary Partitioning». In: *CoRR* abs/1911.04070 (2019). arXiv: 1911.04070. URL: http://arxiv.org/abs/1911.04070 (cit. a p. 29).

Siti web consultati

[1] Shannon AI. *Self Explaining Structures Improve NLP Models*. Accessed: 2024-07-29. 2020. URL: https://github.com/ShannonAI/Self_Explaining_Structures_Improve_NLP_Models.

[2] Wikipedia contributors. *Structural risk minimization — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Structural_risk_minimization&oldid=1198145991. Accessed 2024-07-30. 2024 (cit. a p. 12).

[3] Hugging Face. *Transformers*. Accessed: 2024-08-23. 2024. URL: https://huggingface.co/transformers/index (cit. alle pp. 32, 36).

[4] Hugging Face. *Transformers*. Accessed: 2024-08-26. 2024. URL: https://huggingface.co/datasets/SetFit/sst5 (cit. a p. 32).

[7] Yuxin Jiang. *SST-2-sentiment-analysis*. Accessed: 2024-08-26. 2020. URL: https://github.com/YJiangcm/SST-2-sentiment-analysis.

[8] Thorsten Joachims. *SVM-Light Support Vector Machine*. Accessed: 2024-07-29. 2017. URL: https://www.cs.cornell.edu/people/tj/svm_light/.

[11] Alessandro Moschitti. *SIGIR 2013 Tutorial*. Accessed: 2024-07-29. 2013. URL: https://disi.unitn.it/moschitti/SIGIR-tutorial.htm.

[12] Alessandro Moschitti. *Tree Kernels in SVM-LIGHT*. Accessed: 2024-07-29. 2006. URL: http://disi.unitn.it/moschitti/Tree-Kernel.htm (cit. alle pp. 10, 14).

[14] NLPbox. *stanford-corenlp-docker*. Accessed: 2024-07-29. 2024. URL: https://github.com/NLPbox/stanford-corenlp-docker?tab=readme-ov-file.

[15] paperswithcode. *Sentiment Analysis on SST-5 Fine-grained classification*. Accessed: 2024-07-29. 2022. URL: https://paperswithcode.com/sota/sentiment-analysis-on-sst-5-fine-grained (cit. a p. 3).

[17] Stanford University Natural Language Processing. *Sentiment Analysis*. Accessed: 2024-07-29. 2013. URL: https://nlp.stanford.edu/sentiment (cit. a p. 27).

[20] stanfordnlp. *CoreNLP*. Accessed: 2024-07-29. 2024. URL: https://github.com/stanfordnlp/CoreNLP.