Large Language Models

Architettura, Addestramento e Impatti Applicativi

[Il tuo nome]

Anno Accademico 2024–2025

Indice

Capitolo 1

Introduzione

1.1 Introduzione

Negli ultimi anni, i *Large Language Models* (LLM) hanno rappresentato un punto di svolta nel campo del *Natural Language Processing* (NLP). Questi modelli, basati sull'architettura *Transformer*, hanno raggiunto risultati senza precedenti in una vasta gamma di compiti linguistici, come la traduzione automatica, la generazione di testo e la comprensione semantica.

L'interesse verso gli LLM non deriva solo dalle loro prestazioni tecniche, ma anche dalle implicazioni economiche, etiche e sociali che ne derivano. Oggi, modelli come GPT, LLaMA, PaLM e Claude alimentano sistemi di intelligenza artificiale conversazionale e strumenti di supporto scientifico, didattico e creativo.

1.1.1 Motivazioni

La crescente adozione degli LLM nei contesti accademici e industriali richiede una comprensione profonda delle loro architetture e dei processi di addestramento. Comprendere come questi modelli apprendono, generalizzano e talvolta falliscono è essenziale per garantire un uso responsabile e consapevole dell'IA.

Questa tesi nasce dall'esigenza di:

- analizzare le basi teoriche e pratiche che rendono gli LLM così efficaci;

- valutare le principali tecniche di ottimizzazione e allineamento;
- discutere i limiti, i bias e i rischi connessi al loro impiego.

1.1.2 Obiettivi e struttura della tesi

L'obiettivo generale di questa tesi è esplorare i principi alla base dei Large Language Models, discutendo il loro funzionamento interno, le tecniche di addestramento e le implicazioni derivanti dal loro utilizzo.

La tesi è strutturata come segue:

- **Capitolo 2** – Descrive le basi teoriche e l'architettura *Transformer*, cuore degli LLM moderni.
- **Capitolo 3** – Presenta il processo di *pre-training*, *fine-tuning* e *reinforcement learning from human feedback* (RLHF).
- **Capitolo 4** – Analizza i risultati ottenuti e le metriche di valutazione.
- **Capitolo 5** – Discute le implicazioni etiche, i rischi e le prospettive future dell'uso degli LLM.
- **Capitolo 6** – Riassume le conclusioni e propone possibili sviluppi futuri.

In sintesi, questa tesi intende offrire una panoramica completa degli LLM, integrando l'aspetto tecnico con una riflessione critica sul loro impatto nella società contemporanea.

Capitolo 2

Parole e Tokens

2.1 Parole e Tokens

Viviamo immersi nei simboli: parole, numeri, codici, linguaggi formali. E' necessario trovare un modo per preparare i modelli alla manipolazione di tali simboli, e tale processo è chiamato **tokenizzazione**. Essa consiste nella pratica di mettere insieme dei caratteri per costruire dei mattoncini basilari la cui combinazione di essi ricostruisce un intero simbolo. In questo capitolo verrà introdotto il sitema **Unicode**, il moderno sistema per rappresentare i caratteri e la codifica **UTF-8**. Successivamente verrà presentato un aloritmo standard detto **Byte-Pair Encoding (BPE)** che automaticamente spezza un input testuale in tokens. Tutti i sistemi di tokenizzazione dipendono dalle **Regular Expressions**. Infine si introdurrà una metrica chiamata **Edit Distance** utile a misurare la similarità tra parole o stringhe.

2.1.1 Le parole

Quante parole ci sono nella frase:

> *They picnicked by the pool, then lay back on the grass and looked at the stars.*

Questa frase contiene **16 parole** se non consideriamo la punteggiatura, oppure **18** se la includiamo. La scelta dipende dal compito: i modelli di linguaggio, ad esempio, tendono a considerare la punteggiatura come parole separate, mentre in altri contesti si può ignorarla.

Nel **linguaggio parlato** la definizione di "parola" diventa ancora più com-

plessa. In un'*utterance* (cioè una frase pronunciata), possiamo trovare **disfluenze** come *uh* o *um*, oppure parole interrotte come *main-* in "I do uh main- mainly business data processing". Questi elementi sono detti rispettivamente *filled pauses* e *fragments*. A seconda dell'applicazione, possiamo decidere se trattarli come vere e proprie parole oppure no. Ad esempio, in un sistema di **trascrizione automatica** potremmo volerli rimuovere; ma in un sistema di **riconoscimento vocale**, mantenerli può essere utile, perché le disfluenze forniscono indizi sul ritmo e la struttura del discorso, e possono persino aiutare a identificare il parlante.

Un'altra distinzione importante riguarda i **word types** e le **word instances** (chiamate anche *word tokens*).

- I **word types** sono le **parole distinte** presenti in un corpus: se il vocabolario è V, allora il numero di tipi è la sua dimensione $|V|$.
- Le **word instances** sono invece il **numero totale di parole effettive**, cioè le *running words*, indicate con N.

Nella frase del picnic, ignorando la punteggiatura, abbiamo **14 tipi** e **16 occorrenze** di parole:

They picnicked by the pool, then lay back on the grass and looked at the stars.

Infine, dobbiamo ancora prendere decisioni come: le parole *They* e *they* sono da considerarsi uguali? La risposta dipende dall'obiettivo. In alcuni casi possiamo trattarle come lo stesso tipo di parola, in altri la distinzione tra maiuscole e minuscole è significativa. Una conseguenza di queste definizoni è che maggiore è il numero di parole in un corpus.

La relazione tra numero di tipi e numero di istanze: la legge di Heaps (o di Herdan)

La relazione tra il numero di tipi di parola $|V|$ e il numero totale di parole N segue una legge empirica nota come **Legge di Herdan** (Herdan, 1960) o **Legge di Heaps** (Heaps, 1978). Essa descrive come la dimensione del vocabolario cresce al crescere della lunghezza del testo, ed è espressa dalla seguente equazione:

$$|V| = kN^{\beta} \tag{2.1}$$

dove k e β sono costanti positive e $0 < \beta < 1$. Il valore di β dipende dal tipo di testo e dal genere linguistico: in molti casi empirici varia tra 0.44 e 0.56. In termini qualitativi, possiamo dire che la dimensione del vocabolario cresce un po' più rapidamente della radice quadrata della lunghezza del testo (in parole).

La legge di Heaps riflette il fatto che, man mano che si leggono o generano più parole, si incontrano continuamente termini nuovi. Tuttavia, la crescita del vocabolario rallenta progressivamente: all'inizio ogni nuova parola è spesso inedita, ma col tempo la maggior parte delle parole incontrate sono già apparse in precedenza.

È anche possibile distinguere due categorie di parole:

- le **function words**, ovvero parole grammaticali come *a*, *of*, *the*, il cui numero è tendenzialmente limitato in una lingua;
- le **content words**, cioè sostantivi, verbi e aggettivi che esprimono significato, e che possono crescere indefinitamente (ad esempio nomi propri o termini tecnici).

Modelli che distinguono tra queste due classi di parole possono mostrare due diverse fasi di crescita del vocabolario, con due valori distinti di β: uno iniziale (quando si introducono sia parole grammaticali sia di contenuto), e uno successivo, in cui solo le parole di contenuto continuano ad aumentare.

2.1.2 Unicode

L'Unicode standard è un metodo per rappresentare testo scritto usando qualsiasi carattere in qualsiasi lingua. Un piccolo accenno alla storia. Nel 1960 i caratteri latente usati per scrivere l'inglese weano rappresentati con un codice chiamato **ASCII** (American Standard Case for Information Interchange). Un byte è una sequenza di 8 bit e quindi può rappresentare

$$N = 2^8 = 256$$

valori diversi da `0` a `255` in decimale, o da `00` a `FF` in esadecimale. Lo standard ASCII del 1960 usava solamente 7 bit, quindi 128 valori diversi che andavano da 0 a 127. Il bit più significativo *high-order bit*, quello più a sinistra, veniva impostato a 0 e quindi non veniva utilizzato per codificare caratteri. Perché

solamente 127 caratteri? Dei 128 possibili codici (da 0 a 127) 95 rappresentano **caratteri stampabili** (lettere, numeri, punteggiatura, simboli). 33 (da 0 a 31 e il 127) sono **codici di controllo** non stampabili. Servivano per le teletypes, cioè macchine da scrivere elettroniche e antichi temrinali per indicare cose come andare a capo, nuova linea, cancellare, bip sonoro ecc.

Tipo di codice	Range ASCII	Quantità	Descrizione
Controllo	0–31, 127	33	Istruzioni per dispositivi (non caratteri st
Stampabili	32–126	95	Lettere, numeri, simboli, punteggiatura
Totale	0–127	128	Usano solo 7 bit (bit alto = 0)

Tabella 2.1: Classificazione dei codici ASCII

Quindi il primo bit veniva messo sempre a zero in parte perché non era necessario per i sistemi informatici del tempo, in parte perché questo faceva risparmiare traffico di trasmissione e veniva quindi utilizzato solo come bit di verifica e di controllo per verificare che non vi fossero errori di trasmissione. Solo successivamente si è iniziata a sentire l'esigenza di introdurre ulteriori caratteri come le lettere accentate o ulteriori simboli e di conseguenza anche quindi la necessità di utilizzare il bit rimantente. Per esempio i Cinesi hanno circa 100.000 caratteri nel sistema Unicode e in totale sono circa 150.000. Quindi come si fa ad organizzare questa necessità? In

Code points

Il sistema Unicode assegna a ogni carattere del mondo un identificatore univoco chiamato code point. Un code point è una rappresentazione astratta del carattere (non della sua forma grafica) ed è identificato da un numero, tradizionalmente scritto in esadecimale, che va da `0x0000` a `0x10FFFF` — cioè da 0 a 1.114.111 in decimale. Avere più di un milione di possibili code point significa che c'è ampio spazio per rappresentare tutti i caratteri esistenti — inclusi quelli di lingue con migliaia di simboli, come il cinese (che ne ha circa 100.000), oltre a simboli matematici, emoji e perfino lingue antiche o inventate. Per convenzione i code points si scrivono con il prefisso `U+`. Ad esempio

UTF-8 Encoding

Sebbene il punto di codice (l'ID univoco) sia la rappresentazione Unicode astratta del carattere, non inseriamo semplicemente quell'ID in un file di

Codice Unicode	Carattere	Descrizione
U+0041	A	LATIN CAPITAL LETTER A
U+0061	a	LATIN SMALL LETTER A

Tabella 2.2: Esempi di caratteri Unicode e le loro descrizioni

testo. Invece ogniqualvolta abbimao biosngo di rappresentare un carattere in una stirnga di testo, noi scriviamo un **encoding** del carattere. Ci sono molti metodi di encoding, ma l'UTF-8 è lo standard e l'intero web utlizza tale metodo. I code points vanno da `U+0000` che corrisponde allo 0 a `U+10FFFF` che corrisponde a 1.114.111 nel decimale. Per catturare tutte le possibilità occorrono al minimo **21 bit** questo perché

$$2^{21} = 2.097.152 > 1.114.111$$

Le possibili soluzioni di encoding sono le seguenti

- **UTF-32**. Utilizzare 4 byte (32 bit) per ogni carattere e scriviamo direttamente il numero in un code point

Carattere	Code point	UTF-32 (esadecimale)
h	U+0068	00 00 00 68
e	U+0065	00 00 00 65
l	U+006C	00 00 00 6C
l	U+006C	00 00 00 6C
o	U+006F	00 00 00 6F

Tabella 2.3: Rappresentazione dei caratteri della parola "hello" in UTF-32

  quindi "hello" in UTF-32 occupa 20 byte (5 caratteri x 4 byte). Funziona ma è **poco efficiente**. I file diventano quattro volte più grandi rispetto all'ASCII.

- **UTF-8**. Per risolvere il problema si è inventata una soluzione di encoding a**lunghezza variabile**. I caratteri ASCII (i primi 127 code points, cioè da U+0000 a U+007F) utilizzano **1 solo byte** (esattamente come in ASCII). I caratteri successivi (accenti, simboli, ideogrammi, emoji) usano 2, 3 o 4 byte a a seconda del valore del code point.

  In questo modo "hello" resta identico in ASCII e UTF-8: `68 65 6C 6C 6F`. Ed è per questo che **i file ASCII sono validi anche in UTF-8**.

Carattere	Code point	UTF-8 (esadecimale)	Byte usati
h	U+0068	68	1
e	U+0065	65	1
l	U+006C	6C	1
o	U+006F	6F	1
ñ	U+00F1	C3 B1	2

Tabella 2.4: Esempi di caratteri Unicode e loro codifica UTF-8

Encoding	Byte fissi o variabili?	Byte per carattere	Pro
ASCII	1	1	Semplice, compatibile
UTF-32	Fissi (4)	Sempre 4	Facile da usare in memoria
UTF-16	Variabili (2 o 4)	2–4	Compromesso per lingue eu
UTF-8	Variabili (1–4)	1–4	Efficiente, compatibile con

Tabella 2.5: Confronto tra diverse codifiche di caratteri

Riassumendo:

e in conclusione i code points servono per *identificare logicamente* ogni carattere. Gli encoding (UTF-8, UTF-16 e UTF-32) servono a *rappresentare fisicamente* i code points nei file. UTF-8 è il formato standard del web perché è efficiente, compatibile con ASCII, non introduce byte nulli e può rappresentare tutti i caratteri Unicode.

2.1.3 Subword Tokenization: Byte-Pair Encoding

La **tokenizzazione** è il **primo stadio dell'elaborazione del linguaggio naturale (NLP)**. È il processo di **segmentazione** che converte il testo in **tokens**, ossia unità di base utilizzate dagli algoritmi di elaborazione.

Per convertire un testo in tokens possiamo scegliere tra **parole**, **sillabe** o **caratteri**, ma ognuna di queste opzioni presenta dei limiti. Le parole e le sillabe possono sembrare una buona scelta, ma sono **difficili da definire in modo preciso e coerente** tra lingue e contesti diversi. Le lettere, invece, sono **unità troppo piccole** per catturare il significato linguistico.

In pratica, quindi, si utilizza un **approccio data-driven**, ovvero basato sull'analisi dei dati, per determinare automaticamente le unità linguistiche più appropriate.

Una domanda fondamentale è: **perché abbiamo bisogno di tokenizzare**

l'input? Una ragione principale è che la tokenizzazione consente di convertire il testo in un **insieme deterministico e standardizzato di unità**. In questo modo, diversi sistemi e algoritmi possono operare sullo stesso testo in maniera **coerente e confrontabile**.

Ad esempio, la tokenizzazione ci permette di rispondere in modo univoco a domande come: "Quanto è lungo questo testo?" oppure "'don't' o 'New York' sono un token o due?". Questa **standardizzazione** è quindi essenziale per la **riproducibilità degli esperimenti di NLP** e per il corretto funzionamento di molti algoritmi, come le **misure di perplexity** nei modelli linguistici, che assumono che tutti i testi siano tokenizzati secondo criteri fissi.

Un ulteriore vantaggio dei metodi di **tokenizzazione basati su unità più piccole**, come **morfemi** o **lettere**, è che essi **eliminano il problema delle parole sconosciute** (*unknown words*).

Cosa intendiamo con questo? Nei sistemi di NLP, gli algoritmi apprendono informazioni sul linguaggio da un insieme di testi chiamato **corpus di addestramento** (*training corpus*), e poi applicano queste conoscenze a un **corpus di test**, che contiene testi nuovi. Il problema nasce quando nel corpus di test compaiono **parole mai viste** durante l'addestramento.

Ad esempio, se il sistema ha incontrato le parole *low*, *new* e *newer*, ma non *lower*, non saprà come trattare quest'ultima, poiché non la riconosce come unità nota.

Per affrontare questo problema, i **tokenizzatori moderni** adottano un **approccio data-driven**, inducendo automaticamente **insiemi di token più piccoli delle parole**, chiamati **subword units** o **subwords**. Queste unità possono essere **sottostringhe arbitrarie** oppure **unità linguisticamente significative**, come i morfemi *-er* o *-est*.

Nei moderni schemi di tokenizzazione, molti tokens corrispondono a **parole intere**, ma altri rappresentano **morfemi o frammenti ricorrenti di parole**. Questo approccio consente di rappresentare **qualsiasi parola non vista** come una **combinazione di subword già note**.

Ad esempio, se il sistema non avesse mai incontrato la parola *lower*, potrebbe comunque segmentarla in *low* e *er*, entrambe già presenti nel vocabolario. Nel caso limite, una parola particolarmente rara o un acronimo (come *GRPO*) potrebbe essere **scomposto in singole lettere**, garantendo comunque una rappresentazione coerente e gestibile.

Infine, gli **algoritmi di tokenizzazione** maggiormente utilizzati nei moderni **modelli di linguaggio** sono due:

- **Byte-Pair Encoding (BPE) [sennrich-etal-2016-neural]**
- **Unigram Language Modeling [kudo-2018-subword]**

Come quasi tutti i sistemi di tokenizzazione, il **Byte-Pair Encoding (BPE)** si compone di due parti: un **trainer** e un **encoder**. In generale, nella fase di **training dei tokens** prendiamo un **corpus di addestramento** e da esso estraiamo un **vocabolario**, ovvero un insieme di tokens che rappresentano le unità apprese dal modello. Successivamente, l'**encoder** utilizza questo vocabolario per **segmentare e convertire una nuova frase di test** nei corrispondenti tokens appresi durante la fase di training.

Nel paragrafo seguente vedremo più nel dettaglio come funziona l'algoritmo BPE e come esso costruisce progressivamente il proprio vocabolario di subword.

2.1.4 BPE Training

L'algoritmo di **Byte-Pair Encoding (BPE)** effettua un processo di **fusione iterativa** di tokens adiacenti più frequenti per creare progressivamente tokens più lunghi. In altre parole, il BPE parte da unità molto piccole (come i singoli caratteri) e, ad ogni iterazione, unisce le coppie di simboli più frequenti nel corpus di addestramento, costruendo un vocabolario di subword sempre più ricco.

All'inizio, il vocabolario è semplicemente l'insieme di tutti i **caratteri individuali** presenti nel corpus. L'algoritmo esamina quindi il **corpus di training** e trova la **coppia di simboli adiacenti più frequente**. Immaginiamo, ad esempio, che il nostro corpus sia composto da 10 caratteri e che il vocabolario iniziale contenga 5 simboli: {A, B, C, D, E}.

A B D C A B E C A B

La coppia più frequente è "A B". L'algoritmo quindi unisce questa coppia in un nuovo token **AB**, lo aggiunge al vocabolario e sostituisce tutte le occorrenze di "A B" con "AB":

AB D C AB E C AB

Ora il vocabolario diventa {A, B, C, D, E, AB} e il corpus ha lunghezza 7. La coppia più frequente diventa "C AB", che viene fusa in un nuovo token

CAB, aggiornando il vocabolario a {A, B, C, D, E, AB, CAB} e riducendo ulteriormente la lunghezza del corpus.

L'algoritmo continua in questo modo a **contare e fondere** coppie di tokens, creando sequenze sempre più lunghe, fino a quando non vengono effettuate **k fusioni**. Il parametro **k** rappresenta quindi il numero di nuove unità (subword) da apprendere. Il vocabolario finale sarà costituito dall'insieme dei caratteri iniziali più i **k nuovi simboli generati**. Questo è il **cuore dell'algoritmo BPE**.

In pratica, però, il BPE non viene eseguito sull'intera sequenza di caratteri del corpus, ma solo **all'interno delle parole**. Ciò significa che le fusioni non attraversano i confini tra parole. Per ottenere questo risultato, il corpus viene prima suddiviso in parole utilizzando **spazi bianchi e punteggiatura** (spesso tramite espressioni regolari). In questo modo, ogni parola viene trattata come una sequenza di caratteri indipendente, con le fusioni permesse solo all'interno di ciascuna di esse. Vediamo ora un piccolo esempio sintetico con il seguente corpus:

set new new renew reset renew

Dividendo il corpus in parole (con i rispettivi conteggi di frequenza), otteniamo:

Corpus	**Conteggio**
_ n e w	2
_ r e n e w	2
_ s e t	1
_ r e s e t	1

Il vocabolario iniziale sarà: { _, e, n, r, s, t, w }.

L'algoritmo ora conta tutte le **coppie di simboli adiacenti**. La coppia più frequente è **n e**, che compare 4 volte (due in "new" e due in "renew"). Questa viene quindi fusa in un nuovo simbolo **ne**, aggiornando il vocabolario:

{ _, e, n, r, s, t, w, ne }

Il corpus aggiornato diventa:

_ ne w _ r e ne w _ s e t _ r e s e t

La coppia più frequente ora è **ne w**, che viene fusa nel nuovo token **new**:

_ new _ r e new _ s e t _ r e s e t

Aggiornando così il vocabolario: { _, e, n, r, s, t, w, ne, new }.

Successivamente, la coppia più frequente è **r e**, che viene fusa in **re**, inducendo naturalmente il **prefisso linguistico "re-"**:

_ new _ re new _ s e t _ re s e t

Se continuiamo, le prossime fusioni (*merge*) e i rispettivi vocabolari sono i seguenti:

Merge	**Vocabolario corrente**
(_ , new)	`_, e, n, r, s, t, w, ne, new, _r, _re, _new`
(_re, new)	`_, e, n, r, s, t, w, ne, new, _r, _re, _new, _renew`
(s, e)	`_, e, n, r, s, t, w, ne, new, _r, _re, _new, _renew, se`
(se, t)	`_, e, n, r, s, t, w, ne, new, _r, _re, _new, _renew, se, set`

Infine, il processo di training del BPE può essere riassunto dal seguente pseudocodice:

```
def byte_pair_encoding(corpus, num_merges):
"""
Implementazione semplificata del training BPE.

Parametri:
corpus (list of list of str): corpus tokenizzato a livello di carattere.
Esempio: [["n", "e", "w"], ["r", "e", "n", "e", "w"]]
num_merges (int): numero di merge (k)

Ritorna:
vocab (set): insieme dei token appresi
"""

# 1. vocabolario iniziale: tutti i caratteri unici
```

```
vocab = set(char for word in corpus for char in word)

# 2. funzione ausiliaria per contare coppie adiacenti
def get_stats(corpus):
pairs = {}
for word in corpus:
for i in range(len(word) - 1):
pair = (word[i], word[i + 1])
pairs[pair] = pairs.get(pair, 0) + 1
return pairs

# 3. funzione per unire la coppia più frequente
def merge_pair(pair, corpus):
merged = []
bigram = " ".join(pair)
replacement = "".join(pair)
for word in corpus:
word_str = " ".join(word)
# sostituisce la coppia più frequente con il nuovo token
new_word = word_str.replace(bigram, replacement)
merged.append(new_word.split())
return merged

# 4. ciclo principale: effettua k fusioni
for i in range(num_merges):
pairs = get_stats(corpus)
if not pairs:
break
# coppia più frequente
best_pair = max(pairs, key=pairs.get)
corpus = merge_pair(best_pair, corpus)
vocab.add("".join(best_pair))
print(f"Merge {i+1}: {best_pair} -> {''.join(best_pair)}")

return vocab


# Esempio d'uso
corpus = [
["_", "n", "e", "w"],
["_", "r", "e", "n", "e", "w"],
```

```
["_", "s", "e", "t"],
["_", "r", "e", "s", "e", "t"]
]

vocab = byte_pair_encoding(corpus, num_merges=7)
print("\nVocabolario finale:")
print(vocab)
```

In sintesi, il **BPE training** costruisce un vocabolario di subword apprendendo iterativamente le combinazioni di simboli più frequenti, consentendo così ai modelli linguistici di gestire in modo efficiente parole nuove o rare.

2.1.5 BPE Encoder

In questa fase non si impara pià nulla. Una volta appreso il **vocabolario**, entra in gioco il **BPE encoder**, che viene utilizzato per tokenizzare nuove frasi di test. L'encoder applica al testo di input le stesse **regole di fusione** (*merge rules*) apprese durante la fase di training, seguendo rigorosamente lo **stesso ordine** in cui queste sono state acquisite. In altre parole, l'encoder non tiene conto delle frequenze presenti nei dati di test, ma utilizza esclusivamente le regole derivate dalle frequenze osservate nel corpus di addestramento.

Il processo di codifica avviene come segue: innanzitutto, ogni parola della frase di test viene segmentata nei suoi **caratteri costituenti**. Successivamente, le regole di fusione vengono applicate una dopo l'altra in modo **greedy** (cioè sempre scegliendo la fusione valida più lunga possibile), secondo l'ordine stabilito durante il training. Ad esempio, la prima regola sostituirà ogni occorrenza di `n e` con `ne`, la seconda unirà `ne w` in `new`, e così via. Alla fine del processo, molte delle fusioni ricreeranno semplicemente parole già viste nel corpus di addestramento. Tuttavia, il vantaggio del BPE è che esso impara anche **unità morfologiche riutilizzabili**, come il prefisso *re-* (che potrà apparire in combinazioni non viste come *revisit* o *rearrange*), oppure la radice *new*, che potrà ricomparire in parole nuove come *anew*. Nelle applicazioni reali, il BPE viene eseguito su corpora molto grandi, con **decine di migliaia di fusioni** (ad esempio 50.000, 100.000 o persino 200.000 merge). Il risultato è che la maggior parte delle parole comuni può essere rappresentata come un singolo token, mentre solo le parole più rare o sconosciute vengono suddivise in più subword. Questo approccio funziona particolarmente bene per lingue come l'inglese. Nei sistemi multilingue, invece, il vocabolario appreso può es-

sere fortemente **sbilanciato verso l'inglese**, lasciando un numero inferiore di tokens disponibili per le altre lingue, come approfondiremo più avanti.

Le regole di fusione. Durante la fase di training del BPE non viene prodotto soltanto un vocabolario finale, ma anche un insieme ordinato di **regole di fusione** (*merge rules*). Ogni regola rappresenta una coppia di simboli che è stata fusa durante l'addestramento, e l'ordine in cui le regole vengono apprese è fondamentale.

Il **vocabolario** descrive l'insieme dei tokens appresi, mentre le **regole di fusione** costituiscono la sequenza di operazioni che l'encoder dovrà applicare per tokenizzare nuovi testi. In fase di encoding, infatti, il modello non ricalcola le frequenze nei dati di test, ma applica in modo deterministico le stesse regole apprese nel training, nell'ordine in cui sono state salvate.

In pratica, un sistema BPE salva due file distinti:

- un file di vocabolario (`vocab.txt`), che contiene tutti i tokens finali;
- un file di regole (`merges.txt`), che elenca le coppie fuse in ordine di apprendimento.

Queste informazioni permettono di riprodurre in modo coerente la stessa tokenizzazione su nuovi testi.

2.1.6 Corpora

Le **parole** non compaiono nel vuoto: ogni testo che analizziamo è prodotto da una o più persone, in un determinato **contesto linguistico**, in un luogo e in un tempo specifico, con un preciso scopo comunicativo. Per questo motivo, ogni **corpus linguistico** è una rappresentazione situata del linguaggio, e riflette una varietà di fattori come la lingua, il dialetto, la provenienza geografica, lo stile o il genere testuale.

Un aspetto fondamentale riguarda la **varietà linguistica**. Gli algoritmi di NLP dovrebbero essere testati e sviluppati su più lingue, e non solo sull'inglese, che purtroppo domina gran parte della ricerca contemporanea (Bender, 2019). Anche all'interno di una stessa lingua, esistono molteplici **varietà** e **registri**, legati a differenze regionali, sociali o culturali. Ad esempio, i testi che incorporano caratteristiche dell'**African American English (AAE)** o dell'**African American Vernacular English (AAVE)**

presentano strutture e forme diverse rispetto al Mainstream American English (MAE), e richiedono strumenti NLP in grado di riconoscere e gestire tali varietà (Blodgett et al., 2016; King, 2020).

Inoltre, è frequente che in uno stesso testo compaiano più lingue, fenomeno noto come **code-switching**. Questo è comune in contesti multilingue, ad esempio nei social media, dove un utente può alternare spontaneamente parole o frasi in lingue diverse (Solorio et al., 2014; Jurgens et al., 2017).

Oltre alla lingua, altre dimensioni influenzano la natura di un corpus:

- il **genere testuale** (articoli di giornale, narrativa, testi scientifici, conversazioni, social media, ecc.);
- le **caratteristiche demografiche** degli autori (età, genere, classe sociale, provenienza);
- il **periodo storico** in cui il testo è stato prodotto, poiché la lingua cambia nel tempo.

Poiché il linguaggio è così fortemente situato, è fondamentale, nello sviluppo di modelli di NLP basati su corpora, considerare **chi ha prodotto il linguaggio**, **in quale contesto** e **con quale scopo**. Per garantire trasparenza, riproducibilità e uso etico dei dati, è buona pratica accompagnare ogni corpus con una **datasheet** (Gebru et al., 2020) o una **data statement** (Bender et al., 2021). Questi documenti descrivono in modo dettagliato le caratteristiche e il processo di costruzione del corpus, specificando informazioni fondamentali come:

- **Motivazione:** per quale scopo è stato raccolto il corpus, da chi e con quale finanziamento;
- **Contesto situazionale:** quando e in quali condizioni il testo è stato scritto o parlato (ad esempio: linguaggio spontaneo, social media, dialogo, monologo, ecc.);
- **Varietà linguistica:** quale lingua, dialetto o regione rappresenta il corpus;
- **Demografia degli autori:** età, genere, provenienza socioeconomica o etnica dei produttori del testo;

- **Processo di raccolta:** dimensione del corpus, modalità di campionamento, consenso informato, eventuali pre-processing e metadati disponibili;
- **Annotazione:** tipo di annotazioni presenti, formazione e caratteristiche degli annotatori, metodologia utilizzata;
- **Distribuzione:** eventuali vincoli di copyright o restrizioni di proprietà intellettuale.

L'inclusione di una datasheet o di una data statement consente di **documentare il contesto, la provenienza e i limiti del corpus**, facilitando la valutazione critica dei risultati e promuovendo pratiche di NLP più **trasparenti, eque e riproducibili**.

2.1.7 Minimum Edit Distance

La **Minimum Edit Distance (MED)** è una misura di somiglianza tra due stringhe. Essa rappresenta il *numero minimo di operazioni di modifica* (inserzioni, cancellazioni o sostituzioni) necessarie per trasformare una stringa sorgente X in una stringa obiettivo Y.

Date due stringhe

$$X = x_1, x_2, \ldots, x_n \quad \text{e} \quad Y = y_1, y_2, \ldots, y_m,$$

definiamo $D[i, j]$ come la distanza minima per trasformare il prefisso $X[1..i]$ in $Y[1..j]$. La distanza tra le stringhe intere sarà quindi $D[n, m]$.

La **ricorrenza** del problema, basata sulla programmazione dinamica, è:

$$D[i,j] = \min \begin{cases} D[i-1,j] + \text{del-cost}(x_i) & \text{(cancellazione)} \\ D[i,j-1] + \text{ins-cost}(y_j) & \text{(inserzione)} \\ D[i-1,j-1] + \text{sub-cost}(x_i, y_j) & \text{(sostituzione o match)} \end{cases}$$

con condizioni di base

$$D[0,0] = 0, \quad D[i,0] = i, \quad D[0,j] = j.$$

Nella versione di **Levenshtein**, i costi sono:

$$\text{ins-cost}(x) = 1, \quad \text{del-cost}(x) = 1, \quad \text{sub-cost}(x,y) = \begin{cases} 0 & \text{se } x = y \\ 2 & \text{se } x \neq y \end{cases}$$

L'algoritmo seguente mostra l'implementazione in pseudocodice:

```
function MIN_EDIT_DISTANCE(source, target)
    n ← length(source)
    m ← length(target)
    create matrix D[n+1, m+1]

    for i = 1 to n: D[i,0] ← i
    for j = 1 to m: D[0,j] ← j

    for i = 1 to n:
        for j = 1 to m:
            if source[i] == target[j]:
                cost ← 0
            else:
                cost ← 2
            D[i,j] ← min(
                D[i-1,j] + 1,      # cancellazione
                D[i,j-1] + 1,      # inserzione
                D[i-1,j-1] + cost  # sostituzione
            )
    return D[n,m]
```

Esempio. Consideriamo la trasformazione di `intention` in `execution` con costi $(1, 1, 2)$:

$$D(\texttt{intention}, \texttt{execution}) = 8.$$

Una possibile sequenza di operazioni ottimali è:

$$\begin{aligned}
\texttt{intention} &\xrightarrow{\text{del } i} \texttt{ntention} \\
&\xrightarrow{\text{sub } n\to e} \texttt{etention} \\
&\xrightarrow{\text{sub } t\to x} \texttt{exention} \\
&\xrightarrow{\text{ins } u} \texttt{exentionu} \\
&\xrightarrow{\text{sub } n\to c} \texttt{executiou} \\
&\xrightarrow{\text{sub } o\to n} \texttt{execution}
\end{aligned}$$

Totale: 1 cancellazione, 1 inserzione, 4 sostituzioni $\rightarrow$ costo complessivo $= 8$.

La MED è utilizzata in molti ambiti del Natural Language Processing: *correzione ortografica* (distanza tra parola digitata e dizionario), *riconoscimento*

vocale (Word Error Rate), *traduzione automatica* (allineamento di frasi) e persino in *biologia computazionale* (confronto tra sequenze di DNA).

È importante notare che la Minimum Edit Distance misura una **distanza di forma**, non di significato. Due parole possono essere simili ortograficamente ma diverse semanticamente (*es.* `cane` e `pane`), o viceversa (*es.* `gatto` e `felino`). Come vedremo più avanti con gli **embeddings semantici**, esistono metriche in grado di catturare la *somiglianza di significato*, anziché quella puramente ortografica.

$$\text{Minimum Edit Distance} = \min_{\text{tutte le sequenze di edit}} \sum \text{costo delle operazioni}$$

Essa fornisce una misura quantitativa della somiglianza tra stringhe e rappresenta una base fondamentale per molti algoritmi di NLP basati sulla **programmazione dinamica**.

Capitolo 3

N-gram Language Models

Si consideri la frase:

> Il Po è il fiume più grande della ...

È naturale aspettarsi completamenti come *Lombardia* o *penisola italiana*, mentre parole come *frigorifero* risultano improbabili. Questa intuizione — scegliere la parola più plausibile dato un contesto — è alla base dei *language models* (LM), che stimano la probabilità di una parola successiva $P(w_t \mid w_{1:t-1})$. Tali modelli hanno applicazioni pratiche (correzione ortografica, riconoscimento vocale, predizione di parole) e sono il fondamento anche dei moderni LLM, addestrati tramite *next-word prediction*. In questo capitolo ci concentriamo sugli *n-grammi*, i modelli più semplici e trasparenti: una sequenza di n parole che approssima

$$P(w_t \mid w_{1:t-1}) \approx P(w_t \mid w_{t-n+1:t-1}),$$

assumendo una dipendenza di Markov di ordine $n-1$. Gli n-grammi permettono di introdurre i concetti fondamentali della modellazione del linguaggio: stima delle probabilità dai corpus, valutazione con la *perplexity*, generazione tramite *sampling* e gestione della scarsità di dati con *smoothing*, *interpolation* e *backoff*.

3.1 N-Grams

Si inizi a considerare il compito di calcolare la probabilità

$$P(w \mid h)$$

cioè la probabilità di una parola w dato un contesto (o storia) h. Ad esempio, se $h =$ "L'acqua del Po è così splendidamente" e vogliamo conoscere la probabilità che la prossima parola sia "blu", stiamo cercando:

$$P(\text{blu} \mid \text{L'acqua del Po è così splendidamente}).$$

Un modo diretto per stimare questa probabilità è tramite le frequenze relative osservate in un corpus:

$$P(w \mid h) \approx \frac{C(h\,w)}{C(h)},$$

dove $C(h\,w)$ è il numero di volte in cui la sequenza "h w" compare nel corpus, e $C(h)$ è il numero di volte in cui compare la sola storia h. Tuttavia, anche usando corpora molto grandi, raramente troveremo abbastanza occorrenze per frasi lunghe: il linguaggio è creativo e nuove combinazioni di parole compaiono di continuo. Non possiamo quindi stimare con affidabilità le probabilità di intere frasi basandoci solo sui conteggi.

3.1.1 La regola della catena

Per affrontare il problema, decomponiamo la probabilità con la *chain rule* della probabilità:

$$P(w_{1:n}) = \prod_{k=1}^{n} P(w_k \mid w_{1:k-1}),$$

dove $w_{1:k-1}$ indica la sequenza delle prime $k-1$ parole. In questo modo, la probabilità di una frase può essere calcolata come prodotto di probabilità condizionate di ogni parola dato il contesto precedente. Ma resta una difficoltà: non possiamo stimare in modo affidabile $P(w_k \mid w_{1:k-1})$ per contesti lunghi, poiché essi compaiono raramente nei dati.

3.1.2 L'assunzione di Markov e i modelli n-gram

Per risolvere il problema si introduce l'**assunzione di Markov**: invece di considerare tutta la storia, si approssima la dipendenza considerando solo le ultime $N-1$ parole. Si assume quindi che la *probabilità di una parola dipende solo dalla probabilità delle $N-1$ precedenti.* Ad esempio, un **bigramma** ($N = 2$) approssima:

$$P(w_n \mid w_{1:n-1}) \approx P(w_n \mid w_{n-1}),$$

così che:

$P(\text{blu} \mid \text{L'acqua del Po è così splendidamente}) \approx P(\text{blu} \mid \text{splendidamente})$.

In generale, per un modello n-gram:

$$P(w_n \mid w_{1:n-1}) \approx P(w_n \mid w_{n-N+1:n-1}).$$

Sostituendo questa approssimazione nella regola della catena, otteniamo una stima per la probabilità di un'intera sequenza:

$$P(w_{1:n}) \approx \prod_{k=1}^{n} P(w_k \mid w_{k-N+1:k-1}),$$

che per un bigramma diventa

$$P(w_{1:n}) \approx \prod_{k=1}^{n} P(w_k \mid w_{k-2+1:k-1}) = \prod_{k=1}^{n} P(w_k \mid w_{k-1}).$$

3.1.3 Stima MLE per modelli n-gram

La **stima di massima verosimiglianza (MLE)** assegna la probabilità a una sequenza dividendo i conteggi osservati: la probabilità di una parola dipende dalla frequenza relativa al suo contesto.

$$P(w_n \mid w_{n-1}) = \frac{C(w_{n-1}, w_n)}{C(w_{n-1})} \tag{3.1}$$

Per generalizzare:

$$P(w_n \mid w_{n-N+1:n-1}) = \frac{C(w_{n-N+1:n})}{C(w_{n-N+1:n-1})} \tag{3.2}$$

Queste stime funzionano bene nei casi frequenti, ma soffrono quando i bigrammi o trigrammi sono rari o assenti. Per questo motivo, nei paragrafi successivi verranno introdotte tecniche di *smoothing*.

3.1.4 Calcolo in log-spazio e modelli n-gram estesi

Per evitare problemi di *underflow numerico*, le probabilità nei modelli di linguaggio vengono calcolate in **logaritmo**:

$$\log(p_1 \cdot p_2 \cdot \ldots \cdot p_n) = \log p_1 + \log p_2 + \ldots + \log p_n.$$

Tutte le operazioni di moltiplicazione diventano somme, più stabili e computazionalmente efficienti. Quando i dati lo permettono, si utilizzano modelli con contesti più lunghi (trigrammi, 4-grammi, 5-grammi). Esistono anche dataset su larga scala come Google N-grams o COCA, e tecniche avanzate come gli ∞-gram che usano strutture come *suffix arrays* per gestire n arbitrario.

3.1.5 Set di addestramento, sviluppo e test

Per valutare un modello linguistico si usano tre insiemi distinti:

- **Training set**: per stimare i parametri del modello (conteggi, probabilità).
- **Development set (devset)**: per testare modifiche durante lo sviluppo.
- **Test set**: usato una sola volta per valutazione finale e imparziale.

Un buon test set riflette il dominio applicativo. I modelli non devono mai "vedere" il test set in fase di training: ciò porterebbe a stime artificialmente alte e a *overfitting*.

3.1.6 Perplexity

La **perplessità** misura quanto bene un modello predice un testo. È l'inverso normalizzato della probabilità del test set:

$$\text{Perplexity}(W) = P(w_1, w_2, \ldots, w_N)^{-\frac{1}{N}} .$$

oppure, per modelli n-gram:

$$\text{Perplexity}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i \mid w_{i-n+1:i-1})}} .$$

Per evitare problemi numerici, la formula può essere riscritta in **logaritmo**:

$$\text{Perplexity}(W) = \exp\left(-\frac{1}{N}\sum_{i=1}^{N} \log P(w_i \mid w_{i-n+1:i-1})\right) .$$

Più è bassa la perplessità, più il modello è predittivo.

3.1.7 Perplessità come fattore medio di diramazione

La perplessità può essere interpretata anche come il **fattore medio di diramazione** di una lingua. Il *branching factor* rappresenta il numero medio di parole che possono seguire una determinata parola. Consideriamo un linguaggio artificiale molto semplice con un vocabolario costituito da tre colori:

$$L = \{\text{red}, \text{blue}, \text{green}\}$$

Supponiamo che questo linguaggio sia deterministico e che ogni parola possa seguire qualsiasi altra con probabilità uniforme. Il fattore di diramazione in questo caso è 3. Creiamo ora un modello linguistico probabilistico A, in cui ogni parola segue qualsiasi altra con probabilità $\frac{1}{3}$, basato su un training set con conteggi uguali per i tre colori. Consideriamo il seguente test set:

$$T = \text{red red red red blue}$$

La perplessità del modello A sul test set T sarà:

$$\text{Perplexity}_A(T) = P_A(\text{red red red red blue})^{-\frac{1}{5}} = \left(\frac{1}{3^5}\right)^{-\frac{1}{5}} = 3$$

Ora consideriamo un secondo modello linguistico B, addestrato su un training set in cui la parola "red" è molto più frequente. Le probabilità sono le seguenti:

$$P(\text{red}) = 0.8, \quad P(\text{green}) = 0.1, \quad P(\text{blue}) = 0.1$$

Calcolando la perplessità sullo stesso test set per il modello B otteniamo:

$$\text{Perplexity}_B(T) = P_B(\text{red red red red blue})^{-\frac{1}{5}} = (0.8^4 \cdot 0.1)^{-\frac{1}{5}} = 0.04096^{-\frac{1}{5}} \approx 1.89$$

Anche se il vocabolario contiene sempre tre parole (quindi il branching factor "teorico" resta 3), il modello B è molto più sicuro su quale parola verrà dopo (prevede spesso "red"). Questo rende la sequenza più prevedibile e quindi la perplessità più bassa. La perplessità può quindi essere vista come un **fattore di diramazione pesato**, che tiene conto delle probabilità effettive assegnate dal modello linguistico.

3.1.8 Campionamento da un modello linguistico

Un modo fondamentale per comprendere e visualizzare il comportamento di un modello linguistico è **campionare** da esso.

Campionare da una distribuzione significa scegliere dei valori in modo casuale, proporzionalmente alla loro probabilità. In questo contesto, campionare da un modello linguistico significa **generare frasi** secondo le probabilità apprese dal modello: le frasi più probabili verranno generate più frequentemente, mentre quelle improbabili appariranno raramente.

Questa tecnica fu proposta già nei primi lavori di Shannon (1948) e di Miller e Selfridge (1950). È particolarmente semplice da visualizzare nel caso dei modelli **unigramma**.

Esempio: campionamento da un modello unigramma

Supponiamo di avere un modello unigramma in cui ogni parola ha una certa probabilità di comparire. Possiamo immaginare tutte le parole dell'inglese (o dell'italiano) disposte su un intervallo di probabilità compreso tra 0 e 1, ognuna occupando un sottointervallo proporzionale alla sua frequenza.

La figura seguente (Fig. 3.1) rappresenta visivamente questa distribuzione: scegliendo un numero casuale tra 0 e 1, troveremo l'intervallo in cui cade quel numero e stamperemo la parola corrispondente. Ripetendo il processo finché non si genera un token di fine frase (es. `</s>`), otteniamo una frase campionata.

Figura 3.1: Visualizzazione del campionamento da un modello unigramma. Le parole più frequenti (es. *the*, *of*, *a*) occupano intervalli più grandi, quindi è più probabile che vengano selezionate.

Campionamento da modelli bigramma e oltre

Lo stesso principio si applica ai modelli **bigramma** o **n-gramma** di ordine superiore:

- Si inizia campionando una parola iniziale (es. `<s>`, o secondo le probabilità iniziali).
- Si sceglie poi una seconda parola in base alla distribuzione condizionata sul primo bigramma ($P(w_2 \mid w_1)$).
- Si continua campionando parole successive secondo le probabilità condizionate ($P(w_n \mid w_{n-1})$, o più in generale $P(w_n \mid w_{n-N+1:n-1})$) finché non si genera il token `</s>`.

Questo tipo di generazione è utile non solo per **valutare l'apprendimento del modello**, ma anche come base per applicazioni di *text generation*, *autocomplete* o *chatbot*.

3.2 Smoothing, Interpolation e Backoff

Uno dei problemi principali dei modelli n-gram è la presenza di **zeri**: sequenze di parole possibili che non compaiono nel *training set* ma che possono apparire nel *test set*. In questi casi la stima MLE assegna probabilità zero, con due conseguenze:

1. si sottostima la probabilità di sequenze plausibili;
2. la probabilità di un'intera frase può diventare zero, rendendo impossibile il calcolo della *perplexity*.

Per affrontare questo problema si utilizzano tecniche di **smoothing** o **discounting**, che ridistribuiscono parte della massa di probabilità dagli eventi frequenti a quelli rari o non osservati. Esistono varie strategie, tra cui *Laplace smoothing*, *add-k smoothing*, *interpolation* e *backoff*.

3.2.1 Laplace (Add-One) Smoothing

Il metodo più semplice consiste nell'aggiungere 1 a tutti i conteggi prima della normalizzazione. Per un unigramma:

$$P_{\text{Laplace}}(w_i) = \frac{c_i + 1}{N + V},$$

dove c_i è la frequenza della parola w_i, N il numero totale di token e V la dimensione del vocabolario.

Per i bigrammi:

$$P_{\text{Laplace}}(w_n \mid w_{n-1}) = \frac{C(w_{n-1}, w_n) + 1}{C(w_{n-1}) + V}.$$

Questo metodo garantisce probabilità non nulle, ma introduce una forte distorsione: sequenze molto frequenti vengono "schiacciate" e quelle mai osservate ricevono troppa probabilità. Per questo motivo non è usato nei moderni modelli di linguaggio, ma resta utile come base concettuale.

3.2.2 Add-k Smoothing

Un'estensione del metodo precedente consiste nell'aggiungere non 1 ma una costante k (anche frazionaria):

$$P_{\text{Add-k}}(w_n \mid w_{n-1}) = \frac{C(w_{n-1}, w_n) + k}{C(w_{n-1}) + kV}.$$

La scelta di k viene fatta tipicamente ottimizzando su un *devset*. Sebbene riduca gli effetti negativi dell'add-one, anche questo metodo non si dimostra particolarmente efficace per il *language modeling*.

3.2.3 Interpolazione

Un approccio più robusto è combinare modelli di diverso ordine. Ad esempio, se un trigramma non è mai stato osservato, possiamo stimarne la probabilità tramite il corrispondente bigramma o unigramma. La **linear interpolation** calcola quindi:

$$\hat{P}(w_n \mid w_{n-2}, w_{n-1}) = \lambda_1 P(w_n) + \lambda_2 P(w_n \mid w_{n-1}) + \lambda_3 P(w_n \mid w_{n-2}, w_{n-1}),$$

con $\lambda_1 + \lambda_2 + \lambda_3 = 1$.

I pesi λ non sono fissati a priori, ma appresi da un *held-out set*, scegliendo quelli che massimizzano la probabilità dei dati di validazione. Questo metodo consente di sfruttare al meglio le informazioni disponibili, bilanciando specificità e robustezza.

3.2.4 Backoff e Stupid Backoff

In alternativa all'interpolazione, si può ricorrere al **backoff**: se un n-gramma non è disponibile, si "retrocede" a un $(n-1)$-gramma, e così via fino agli unigrammi. Nei modelli classici, per garantire una distribuzione di probabilità corretta, si effettua una forma di *discounting*.

Una variante molto usata in applicazioni pratiche è lo **stupid backoff**. Qui non si cerca di mantenere una distribuzione valida, ma si applica una semplice regola:

$$S(w_n \mid h) = \begin{cases} \dfrac{C(hw_n)}{C(h)} & \text{se } C(hw_n) > 0, \\ \lambda\, S(w_n \mid h') & \text{altrimenti,} \end{cases}$$

dove:

- $S(w_n \mid h)$ è la **funzione di punteggio** (*score*) assegnata alla parola w_n dato il contesto h; non rappresenta una probabilità normalizzata, ma una quantità proporzionale alla probabilità stimata;
- $C(\cdot)$ indica i conteggi osservati nel corpus;
- h' è il contesto accorciato (ad esempio, passando da trigramma a bigramma, o da bigramma a unigramma);
- λ è un fattore di penalizzazione costante (tipicamente 0.4) applicato ogni volta che si "retrocede".

Lo *stupid backoff* non definisce una vera distribuzione probabilistica (poiché i valori di S non sommano a 1), ma fornisce punteggi comparabili tra diverse sequenze. È molto efficace su grandi corpus e in scenari di ricerca su larga scala, come nel motore linguistico di Google [**Brants2007**].

3.3 Perplessità ed Entropia

Abbiamo introdotto la **perplessità** come misura di valutazione per i modelli n-gram. In realtà essa nasce dal concetto di **entropia** in teoria dell'informazione, che fornisce il legame con la **cross-entropia**. Per distinguere correttamente i concetti, introduciamo alcune definizioni fondamentali:

- **Vocabolario** (V): l'insieme finito delle parole (token) conosciute dal modello. Ad esempio, in un corpus ridotto di italiano potremmo avere

$$V = \{\text{gatto}, \text{cane}, \text{dorme}, \text{corre}\}.$$

- **Linguaggio** (L): l'insieme di tutte le *sequenze di parole* costruibili a partire da V. Ad esempio:

$$L = \{\text{gatto dorme},\ \text{cane corre},\ \text{gatto corre}, \ldots\}.$$

  In generale L è potenzialmente infinito, poiché le sequenze possono avere lunghezza arbitraria.

- **Sequenze di parole** ($w_{1:n}$): una specifica frase di lunghezza n, cioè una realizzazione concreta di elementi di V. Per esempio $w_{1:3} = (\text{il},\ \text{gatto},\ \text{dorme})$.

Quando in formule di entropia compare una somma del tipo

$$\sum_{w_{1:n} \in L} p(w_{1:n}) \log p(w_{1:n}),$$

il simbolo L indica l'insieme di tutte le sequenze possibili di lunghezza n (non il solo vocabolario). In altre parole, mentre V è l'insieme statico delle parole, L rappresenta l'insieme dinamico di tutte le frasi che la lingua può generare.

3.3.1 Entropia

L'entropia di una variabile casuale X con distribuzione $p(x)$ è definita come:

$$H(X) = -\sum_{x \in \chi} p(x) \log_2 p(x).$$

Intuitivamente, rappresenta il numero medio di bit necessari a codificare un'informazione in modo ottimale. Ad esempio, se tutti gli eventi sono equiprobabili ($p(x) = 1/|\chi|$), l'entropia diventa $\log_2 |\chi|$.

3.3.2 Entropia di una sequenza

Finora abbiamo considerato l'entropia di una singola variabile casuale (ad esempio, una parola). Per un linguaggio naturale, tuttavia, è più utile ragionare su **sequenze di parole**. Sia dunque $W = (w_1, w_2, \ldots, w_n)$ una frase di lunghezza n. L'entropia della sequenza è definita come:

$$H(w_1, \ldots, w_n) = -\sum_{w_{1:n} \in L} p(w_{1:n}) \log p(w_{1:n}),$$

dove L rappresenta l'insieme di tutte le sequenze possibili di lunghezza n in una lingua. In altre parole, consideriamo tutte le frasi di lunghezza n, ne valutiamo la probabilità e ne calcoliamo l'informazione media.

Tasso di entropia. Poiché l'entropia cresce con la lunghezza della sequenza, conviene normalizzarla per il numero di parole. Si definisce quindi il **tasso di entropia** (o entropia per parola) come:

$$H(L) = \lim_{n \to \infty} \frac{1}{n} H(w_1, \ldots, w_n).$$

Questo valore rappresenta la quantità media di informazione (in bit) che ogni parola porta con sé in una lingua.

Il teorema di Shannon–McMillan–Breiman. Questo teorema dice che, se la lingua può essere vista come un processo stocastico stazionario (le regole non cambiano nel tempo) ed ergodico (osservando a lungo una sequenza, si vedono tutte le probabilità "vere"), allora: non è necessario calcolare la somma su tutte le frasi possibili e l'entropia può essere stimata osservando una sola sequenza molto lunga. Formalmente:

$$H(L) = \lim_{n\to\infty} -\frac{1}{n} \log p(w_1, \ldots, w_n).$$

Intuizione. Il risultato dice che se prendiamo un testo sufficientemente lungo, la media della sorpresa per parola (cioè $-\log p(w_i)$) si stabilizza, e tale valore coincide con l'entropia della lingua. In questo modo l'entropia di un linguaggio diventa un concetto misurabile a partire da testi reali, senza dover enumerare tutte le possibili frasi.

3.3.3 Cross-Entropy

Finora abbiamo supposto di conoscere la distribuzione reale p che genera i dati (cioè la "vera lingua"). Nella pratica, però, non conosciamo mai p: possiamo solo stimarla tramite un modello m (ad esempio, un modello n-gram).

In questi casi si usa la **cross-entropy**, che misura quanto bene il modello m approssima la distribuzione reale p:

$$H(p, m) = \lim_{n\to\infty} -\frac{1}{n} \sum_{W\in L} p(W) \log m(W).$$

Qui stiamo calcolando l'informazione media rispetto alla distribuzione vera p, ma valutata con le probabilità fornite dal modello m. In pratica, chiediamo: *se i dati sono generati da p, quanto "costa" codificarli usando il modello m?*

Stima su sequenze lunghe. Grazie al teorema di Shannon–McMillan–Breiman, non serve considerare tutte le possibili frasi: basta una sequenza sufficientemente lunga. Per un testo di N parole possiamo stimare:

$$H(W) = -\frac{1}{N} \log m(w_1, \ldots, w_N).$$

Relazione con l'entropia. Per definizione, la cross-entropy è sempre maggiore o uguale all'entropia vera:

$$H(p) \leq H(p, m).$$

- Se il modello m coincide con la distribuzione reale p, allora $H(p, m) = H(p)$.
- Se il modello è impreciso, la cross-entropy è più grande, perché m "spende" più bit del necessario per descrivere i dati.

Interpretazione intuitiva. Immaginiamo di voler comprimere un testo in italiano. - Se usassimo la vera distribuzione p della lingua, otterremmo la massima compressione possibile, pari all'entropia $H(p)$. - Usando invece un modello approssimato m (ad esempio un modello bigramma), la compressione sarà meno efficiente: in media useremo più bit per parola. La cross-entropy misura esattamente questo "surplus" di informazione richiesto da un modello approssimato rispetto alla distribuzione ideale.

3.3.4 Relazione con la Perplessità

La **perplessità** è una misura derivata direttamente dalla cross-entropy. Ricordiamo che la cross-entropy $H(W)$ ci dice quanti bit di informazione, in media, sono necessari per codificare una parola di un testo usando un modello linguistico P.

Se l'entropia è misurata in bit, allora $2^{H(W)}$ ha un'interpretazione molto intuitiva: rappresenta il **numero medio di alternative equiprobabili** che il modello deve considerare a ogni passo.

Intuizione. - Se $H(W) = 1$, significa che in media servono 1 bit per parola: il modello è incerto come se dovesse scegliere tra 2 alternative equiprobabili (come il lancio di una moneta). - Se $H(W) = 3$, servono 3 bit per parola: il modello è incerto come se dovesse scegliere tra 8 alternative equiprobabili

(come un dado a 8 facce). - Se $H(W) = 2.585$, il modello è incerto come se avesse circa 6 alternative ugualmente probabili: questo è il caso del dado a 6 facce equo.

Per questo motivo la perplessità viene definita come:

$$\text{Perplexity}(W) = 2^{H(W)}.$$

Forma pratica. Sostituendo l'espressione della cross-entropy, otteniamo:

$$\text{Perplexity}(W) = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log P(w_i \mid w_{i-n+1:i-1})\right).$$

Questa scrittura usa il logaritmo naturale: l'esponenziale ricostruisce il "numero equivalente di scelte equiprobabili".

Interpretazione. La perplessità può quindi essere letta come un **fattore medio di diramazione**: quante possibilità il modello considera plausibili per ogni parola. Un modello con bassa perplessità è meno "perplesso": ha meno incertezza e si avvicina di più a catturare la vera distribuzione della lingua.

3.3.5 Entropia e Compressione dei Dati

Il concetto di entropia non è solo teorico, ma ha applicazioni dirette in informatica, in particolare nella **compressione dati**. Shannon ha dimostrato che l'entropia di una sorgente è il **limite inferiore** del numero medio di bit necessari per rappresentarne i simboli senza perdita di informazione. In altre parole, nessun algoritmo di compressione potrà mai scendere sotto l'entropia: è un vincolo fondamentale.

Codici a lunghezza fissa. Se si rappresentano i simboli con lo stesso numero di bit, non si tiene conto delle probabilità. Ad esempio, il codice ASCII assegna sempre 8 bit per carattere, anche se alcune lettere sono molto più frequenti di altre. Questo approccio è semplice, ma inefficiente.

Codici a lunghezza variabile. Per avvicinarsi al limite teorico dell'entropia, si usano codici che tengono conto della distribuzione di probabilità:

- **Codifica di Huffman**: assegna sequenze di bit più corte ai simboli frequenti e più lunghe a quelli rari. È usata in algoritmi di compressione come `ZIP`, `JPEG`, `MP3`.
- **Arithmetic coding**: rappresenta l'intera sequenza come un unico numero reale in $[0, 1)$. È ancora più efficiente di Huffman ed è usata in standard moderni di compressione come `H.264` e `HEVC`.

Esempio intuitivo. Nel linguaggio naturale, non tutte le lettere hanno la stessa probabilità (in inglese la lettera `e` è molto più frequente di `z`). Una codifica ottimizzata può sfruttare questa distribuzione, riducendo il numero medio di bit per carattere. Ad esempio, mentre l'ASCII usa 8 bit per ogni carattere, l'entropia stimata per l'inglese è circa 1.5 bit per carattere. I moderni algoritmi di compressione testuale si avvicinano a questo limite, ottenendo in pratica circa 2–3 bit per carattere.

In sintesi, l'entropia non solo fornisce una misura dell'incertezza, ma stabilisce anche il **limite teorico della compressione senza perdita**. La perplessità nei modelli linguistici è quindi strettamente connessa a questo concetto: modelli migliori riducono l'incertezza e, di conseguenza, la quantità di informazione necessaria a descrivere un testo.

Capitolo 4

Regressione Logistica e classificazione del testo

Introduzione

4.1 Machine learning e classificazione

Lo scopo della **classificazione** è quello di prendere una singola **osservazione**, estrarre da essa alcune proprietà utili, chiamate **feature**, e assegnarla a una delle **classi** presenti in un insieme discreto di categorie.

Ad esempio, una *task* di classificazione potrebbe consistere nel determinare se una **email** (l'osservazione), descritta dalle relative **feature** (come le parole contenute nel testo, l'orario di invio o il mittente), appartenga alla classe *spam* o *non spam*.

Il modo più efficace per affrontare questo tipo di problema è utilizzare un approccio di **apprendimento supervisionato**, ovvero un paradigma in cui al modello vengono mostrati esempi già etichettati. In questo modo, la macchina impara a generalizzare la funzione sottostante che associa le **feature** ($x \in X$) alle **classi** ($y \in Y$).