



UNIVERSITÀ DEGLI STUDI DI MILANO  
FACOLTÀ DI SCIENZE E TECNOLOGIE

*Corso di Laurea Magistrale in Informatica*

Uso di un modello BERT per la  
correzione di errori generati dal processo  
di OCR su dati testuali

**Relatore:** Prof. Alfio FERRARA

**Correlatore:** Dr. Francesco PERITI

**Autore:** Sebastiano Caccaro

**Matricola:** 958683

Anno Accademico 2020-2021



*”Everything’s in order in a black hole”*  
*Arctic Monkeys*



# Sommario

Ad oggi, sempre più libri cartacei, riviste e giornali presenti in biblioteche e archivi storici stanno venendo trasformati in versioni elettroniche che possono essere manipolate da un computer. A questo scopo, nel corso degli anni sono state sviluppate tecnologie di Optical Character Recognition (comunemente abbreviato con OCR) per tradurre le scansioni e immagini di documenti testuali in testo interpretabile e processabile da un computer. Questi sistemi, però, non sono perfetti e possono introdurre errori nel testo, che possono abbassare drasticamente la precisione di vari task di NLP. In questa tesi si propone un approccio modulare per la correzione di tali errori basato su BERT, un modello di machine learning finalizzato al NLP rilasciato da Google nel 2018. L'approccio proposto è valutato su un dataset creato ad hoc, contenente testi con diverse tipologie e intensità di errore. I risultati sperimentali dimostrano la fattibilità di utilizzare approcci basati su BERT per la correzione di errori in testi acquisiti tramite OCR, mostrando inoltre come ci siano ulteriori margini per migliorare l'approccio proposto.



# Ringraziamenti

*Innanzitutto, vorrei ringraziare il Prof. Ferrara, relatore di questa tesi, e il mio correlatore, il Dott. Periti, che mi hanno guidato egregiamente nello svolgimento di questo elaborato, nonostante le difficoltà derivate dal non potersi incontrare di persona.*

*Un ringraziamento speciale alla mia famiglia, in particolare a mia madre e mio padre, che non mi hanno fatto mai mancare il supporto e i mezzi per portare a termine gli studi.*

*Un super ringraziamento va al Pav, sempre presente nel bene e nel male, con la pioggia e con la neve, con i tuoni e con fulmini, ma non esageriamo sennò si monta la testa.*

*Un ringraziamento speciale va anche ai SUONI aka Dido, Bajo e Vitto perchè alla fine, se porgi il bicchiere per fare cin-cin, sicuramente all'altra persona parte il riflesso di farlo anche lui.*

*Impossibile non ringraziare anche la Marghe per le margheritate durante e dopo la Via degli Dei, e la Greta, che mi aggiorna sempre sulle condizioni di Davide.*

*Ringrazio inoltre il Jack, i  $\delta$ -cappuccio, Roberto Pirelli e i compagni della puglia, e tutti gli amici che, anche se indirettamente, hanno contribuito a non farmi andare in esaurimento prima, durante e dopo il lockdown.*

*Ringrazio infine Filippo, anche se probabilmente ha di meglio da fare.*





# Indice

	iii
<b>Sommario</b>	<b>v</b>
<b>Ringraziamenti</b>	<b>vii</b>
<b>Introduzione</b>	<b>1</b>
<b>1 Stato dell'Arte</b>	<b>5</b>
1.1 OCR Post-processing . . . . .	5
1.1.1 Approcci basati su n-grams . . . . .	6
1.1.2 Approcci basati su NMT . . . . .	8
1.1.3 Approcci basati su BERT . . . . .	11
<b>2 Dataset e Perturbazione</b>	<b>13</b>
2.1 Introduzione . . . . .	13
2.2 Creazione del dataset . . . . .	15
2.3 Perturbazione . . . . .	20
2.3.1 Moduli di perturbazione . . . . .	20
2.3.2 Moduli speciali . . . . .	25
2.3.3 Pipeline . . . . .	26
2.3.4 Superpipeline . . . . .	27
2.4 Configurazione . . . . .	29
2.4.1 Pipeline e Superpipeline . . . . .	29
2.4.2 Dataset . . . . .	32
<b>3 Metodologia di correzione</b>	<b>35</b>
3.1 Introduzione . . . . .	35
3.2 Architettura del sistema . . . . .	38
3.2.1 Panoramica Generale . . . . .	38
3.2.2 BERT Masked Language Modeling . . . . .	40

3.2.3	Modulo di correzione Token . . . . .	41
3.2.4	Modulo di correzione Split . . . . .	46
<b>4</b>	<b>Test e Risultati</b>	<b>53</b>
4.1	Introduzione . . . . .	53
4.2	Metriche . . . . .	54
4.3	Risultati . . . . .	58
<b>5</b>	<b>Analisi dell'errore</b>	<b>67</b>
5.1	Introduzione . . . . .	67
5.2	Creazione di un dataset specifico . . . . .	68
5.3	Analisi generazione dei candidati . . . . .	71
5.4	Analisi scelta dei candidati . . . . .	74
	<b>Conclusioni</b>	<b>81</b>
	<b>Bibliografia</b>	<b>85</b>

# Introduzione

Ad oggi, sempre più libri cartacei, riviste e giornali presenti in biblioteche e archivi storici stanno venendo trasformati in versioni elettroniche che possono essere manipolate da un computer. A questo scopo, nel corso degli anni sono state sviluppate tecnologie di Optical Character Recognition (comunemente abbreviato con OCR) per tradurre le scansioni e immagini di documenti testuali in testo interpretabile e processabile da un computer. Questi sistemi, però, non sono perfetti e possono introdurre errori nel testo. Può accadere, infatti, che durante il processo di scansione alcuni caratteri vengano letti in modo errato, altri vengano aggiunti e altri ancora non riconosciuti. È, ad esempio, particolarmente probabile che caratteri o sequenze di caratteri graficamente simili come "li" e "n" vengano scambiati fra di loro[1]. La frequenza di tali errori è influenzata da fattori quali la condizione di deterioramento di un documento e la qualità di acquisizione dell'immagine[2]: la presenza di granelli di polvere, caratteri scoloriti, pagine ingiallite o artefatti risultanti dalla scansione, ad esempio, influiscono negativamente sulle performance dei sistemi OCR.

La presenza di tali errori in corpora acquisiti tramite OCR risulta problematica in quanto rende meno precisi task di Natural Language Processing (NLP) come, ad esempio, l'esecuzione di query [3] o il topic modelling[4]. Per ovviare a tali problemi, sono state sviluppate varie soluzioni che mirano a minimizzare il quantitativo di errori presenti nel testo estratto. È possibile classificare queste soluzioni nelle seguenti due categorie:

- **OCR Pre-processing:** ricadono in questa categoria tutte quelle tecniche che mirano ad ottenere migliori risultati dall'estrazione del testo attraverso il miglioramento dell'input, ovvero delle immagini, che viene usato dai software di OCR. Tali metodi includono, ma non si limitano a, l'uso di migliori tecniche di scansione, la correzione del contrasto nell'immagine[5] e la rotazione e correzione di deformazioni nell'immagine[6] (Figura 1).
- **OCR Post-processing:** ricadono in questa categoria tutte quelle tecniche che mirano ad individuare e correggere gli errori presenti nell'output generato

dai vari software di OCR. Essendo l'OCR Post-processing oggetto di questa tesi, sarà approfondito a parte nella Sezione 1.1.

OCR pre-processing e post-processing sono spesso usati in congiunzione per ottenere migliori risultati dall'estrazione del testo.

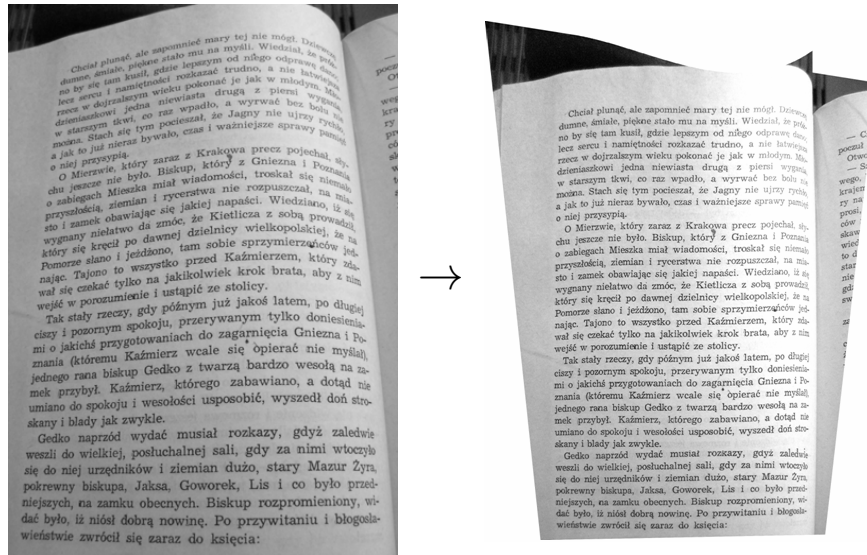


Figura 1: A sinistra foto di una pagina contenente del testo. A destra, foto della stessa pagina pre-processata per facilitare l'estrazione del testo. Esempio preso da [6].

**Scopo e organizzazione della tesi** Lo svolgimento di questa tesi è consistito nello sviluppo di un sistema di OCR post-processing basato su BERT, un modello di machine learning finalizzato al Natural Language Processing sviluppato da Google. Lo sviluppo di tale sistema ha necessitato della creazione di un dataset apposito, nonché di un'adeguata metodologia di test. Il codice prodotto, che permette di replicare i risultati illustrati in questa tesi, è disponibile al seguente link:

[https://github.com/sebacaccaro/codice\\_tesi](https://github.com/sebacaccaro/codice_tesi)

La tesi è organizzata nei seguenti capitoli:

1. **Stato dell'Arte:** in questo capitolo è definito in modo più preciso il problema dell'OCR post-processing, ed è presentata la discussione della letteratura.

2. **Dataset e Perturbazione:** in questo capitolo si discutono le ragioni che hanno portato alla creazione di un dataset apposito, e si descrive la metodologia con la quale tale dataset è stato creato.
3. **Metodologia di correzione:** in questo capitolo si descrive la metodologia implementata dal sistema di correzione sviluppato.
4. **Test e Risultati:** in questo capitolo è descritta la metodologia di test utilizzata, insieme alle metriche definite per la valutazione del sistema di correzione. Sono poi riportati e commentati i risultati dei test eseguiti.
5. **Analisi dell'errore:** in questo capitolo si analizzano le performance dei vari stadi del sistema di correzione, con lo scopo di capire quali fasi sono ottimizzate e quali invece hanno margini di miglioramento.

Sono inoltre riportate in chiusura le conclusioni tratte dai risultati e dall'analisi dell'errore, insieme ad alcuni possibili sviluppi futuri del sistema di correzione.



# Capitolo 1

## Stato dell'Arte

In questo capitolo si passa in rassegna lo stato dell'arte riguardante il tema dell'OCR post-processing.

Nella Sezione 1.1 sono trattati gli approcci più recenti al problema dell'OCR post-processing presenti in letteratura, divisi in base al tipo di metodologia adottata. Per ogni categoria si fornisce una breve introduzione atta a rendere più chiari gli approcci presentati.

### 1.1 OCR Post-processing

In letteratura sono presenti numerosi approcci al problema dell'OCR post-processing, molti dei quali adottano strategie molto differenti. Dato ciò, non è possibile delineare una metodologia generale che ogni approccio segue, ma, in generale, ogni approccio deve:

1. **Identificare gli errori** (Error Detection), ovvero delimitare tutte le sezioni contenenti errori nel testo, senza delimitare sezioni corrette.
2. **Correggere gli errori** (Error Correction), ovvero ripristinare il testo originale nelle sezioni individuate in precedenza

Nelle seguenti sottosezioni sono quindi esposti alcuni dei principali approcci presenti in letteratura, raggruppati nelle seguenti categorie di metodologie:

- Approcci basati su n-grams
- Approcci basati su NMT
- Approcci basati su BERT

### 1.1.1 Approcci basati su n-grams

Per discutere gli approcci basati sugli n-grams è prima necessario definire i concetti di token, tokenizzazione e n-gram.

**Token** È riportata la definizione fornita in [7]: "Un token è una stringa di caratteri contigui compresi fra due spazi, o fra uno spazio e un segno di punteggiatura. Sono token anche [numeri] interi, [numeri] reali o numeri contenenti i due punti (ore, ad esempio 2:00). Tutti gli altri simboli sono considerati essi stessi dei token, eccetto gli apostrofi e i punti di domanda attaccati ad una parola (senza spazi), che in molti casi rappresentano acronimi o citazioni."

Più informalmente è possibile associare il concetto di token a quello di parola nel linguaggio naturale.

**Tokenizzazione** Data la precedente definizione di token, per tokenizzazione si intende il dividere un testo, una frase, o più in generale una stringa, nei token che la compongono. Data quindi una frase  $f \in F$ , tokenizzare una frase vuol dire applicare una funzione:

$$Tok : F \rightarrow T \quad (1)$$

dove ogni  $t \in T$  è una lista  $[t_1, \dots, t_n]$  in cui ogni  $t_i$  è un token appartenente alla frase iniziale. Più informalmente quindi, la tokenizzazione restituisce le singole parole appartenenti alla frase iniziale. Ad esempio, data la frase  $f_{es}$ :

*"Cantami, o Diva, del pelide Achille l'ira funesta che infiniti addusse lutti agli Achei"*

la sua versione tokenizzata  $Tok(f_{es})$  è:

$["Cantami", ",", "o", "Diva", ",", "del", "pelide", "Achille", "l'", "ira", "funesta", "che", "infiniti", "addusse", "lutti", "agli", "Achei"]$

**n-gram** Un n-gram è una sottosequenza contigua di n elementi di una data sequenza [8]. Gli elementi in questione possono essere fonemi, sillabe, lettere parole ecc. Nel proseguo di questo documento ogni riferimento a n-gram, salvo indicazione contraria, si riferisce a n-gram di token. Gli n-gram trovano ampio uso nel campo del NLP, dove sono usati, ad esempio, per creare modelli linguistici statistici.

In seguito è mostrato un esempio di scomposizione di una frase in n-gram di lunghezza 3, detti quindi 3-gram o trigrams. Data la frase del precedente esempio  $f_{es}$ , e data la sua scomposizione in token  $Tok(f_{es})$ , i trigrams formati sono i seguenti:



["Cantami", ",", "o"], [",", "o", "Diva"], ["o", "Diva", ",", "Diva", ",", "del"], [",", "del", "pelide"], ["del", "pelide", "Achille"], ["pelide", "Achille", "l"], ["Achille", "l", "ira"], ["l", "ira", "funesta"], ["ira", "funesta", "che"], ["funesta", "che", "infiniti"], ["che", "infiniti", "addusse"], ["infiniti", "addusse", "lutti"], ["addusse", "lutti", "agli"], ["lutti", "agli", "Achei"]

**Approcci basati su n-grams** Gli approcci descritti in questa sezione utilizzano modelli linguistici basati su n-gram per individuare e correggere gli errori. Le soluzioni proposte in questa sezione fanno entrambe uso del Google Web 1T 5-gram dataset[9], che da qui in poi verrà riferito come GW5. GW5 è un dataset contenente n-grams in lingua inglese (da unigrams a 5-grams) associati alla loro frequenza osservata su un totale di 1 trilione di parole. Tutti gli n-grams sono stati estratti attraverso il crawling di pagine web. L'enorme scala del database e la metodologia tramite la quale è stato ottenuto comporta che sia possibile estrarre da GW5 un ampio lessico che può essere affabilmente usato per fare error detection. Per lo stesso motivo, il dataset si presta bene anche all'applicazione in campi con terminologie di nicchia o altamente specifiche.

Il primo approccio trattato quello presentato in [10]. L'approccio è diviso di tre fasi:

1. Error Detection: sono usati gli unigram in GW5 per identificare gli errori. Ogni token all'interno del testo da correggere non presente nella lista degli unigram è considerato un errore. È quindi chiaro come questo metodo riesca ad individuare (e quindi correggere) solo i non-word errors, ovvero tutti quelli errori che risultano in parole non presenti in un dato lessico. Non sono trattati da questo approccio i real-word errors, ovvero tutti quelli errori che risultano in una parola presente in un dato lessico. Si pensi ad esempio alla parola "sale" interpretata come "sala" da un software OCR.
2. Candidate Spelling Generation: per ogni errore si produce una lista di parole candidate per la correzione. Per fare ciò si scompone la parola errata in 2-grams a livello di carattere. Ad esempio, la parola "sangle" è scomposta in "sa", "an", "ng", "gl", "le". Per ognuna delle parole nel lessico di unigram, in seguito, si conta quante occorrenze dei 2-gram della parola da correggere sono contenute. Ad esempio, la parola "single" contiene tre occorrenze ("ng", "gl", "le"). Le prime 10 parole con più occorrenze dei 2-gram sono considerate i candidati per la correzione.
3. Error Correction: si considera il 5-gram terminante con la parola errata  $[t_1, t_2, t_3, t_4, err]$ . Per ognuno dei candidati  $c_i$  è prodotto il 5-gram  $[t_1, t_2, t_3, t_4, c_i]$ : di questi 5-gram prodotti, quello con più occorrenze all'interno di GW5 contiene la correzione da applicare.

Un approccio simile è esposto in [11]. A differenza dell'approccio precedente, l'error detection e la generazione dei candidati non usano GW5, ma sono usate altre tecniche che mirano a correggere anche i real-word errors. L'approccio utilizzato per l'error correction invece sfrutta lo stesso principio di [10], ma con una logica leggermente più complessa. Il funzionamento è il seguente:

1. Error Detection e Candidate Generation: per i non-word errors GNU-Aspell[12] è utilizzato per individuare gli errori e proporre i possibili candidati per la correzione. Per i real-word errors invece, sono usati dei confusion-set pre-definiti per individuare i possibili errori e generare i candidati. Un confusion-set non è altro che un insieme di parole simili che possono essere confuse fra di loro, come {they' re, their, there}. I candidati per un possibile errore sono quindi le parole appartenenti al suo confusion set.
2. Error Correction. Per ogni candidato si considera un intorno di 2 parole, andando così a comporre un 5-gram. Se il 5-gram in cui è presente l'errore nel testo è  $[t_1, t_2, err, t_3, t_4]$ , allora per il candidato  $c_i$  sarà  $[t_1, t_2, c_i, t_3, t_4]$ . Il candidato scelto è quello il cui 5-gram compare più volte in GW5. In caso nessun 5-gram compaia nel dataset, il processo si ripete con il 3-gram  $[t_2, c_i, t_3]$  e successivamente solo con l'unigram  $[c_i]$ , ovvero viene scelto il candidato con la maggior frequenza nel corpus.

Gli approcci descritti fino ad ora, seppur molto efficaci contro i non-word errors, non correggono o sono poco efficaci contro i real-word errors e altri tipi di errori. Si pensi ad esempio a situazioni in cui token viene separato da spazi, o in cui due token sono fusi insieme. Inoltre, questi approcci richiedono un'elevata quantità di dati per funzionare efficacemente (GW5 occupa 87GiB su disco), il che non li rende facilmente applicabili.

### 1.1.2 Approcci basati su NMT

**Neural Machine Translation** Il Neural Machine Translation (NMT) è un approccio al machine translation che consiste nella costruzione di un solo neural network che può essere messo a punto singolarmente per massimizzare le performance di traduzione[13]. I modelli più recenti nel campo del NMT sfruttano la cosiddetta architettura encoder-decoder (Figura 2). Tali modelli sono anche comunemente riferiti come modelli sequence-to-sequence (seq2seq), in quanto trasformano una sequenza di caratteri o lettere appartenente a un dominio (ad esempio quello delle frasi in lingua italiana) ad una sequenza appartenente ad un altro dominio (ad esempio quello delle frasi in lingua inglese). In questi modelli, la frase iniziale passa attraverso l'encoder, che ne restituisce una rappresentazione sotto forma di

una lista di valori numerici. Successivamente, questi valori sono dati in input al decoder, che produce una frase con lo stesso significato, ma con il vocabolario e la grammatica della lingua target. Encoder e decoder possono essere implementati con diverse architetture, ma, data la natura dei dati trattati (dati sequenziali, come frasi), sono spesso implementati con dei Recurrent Neural Network (RNN).

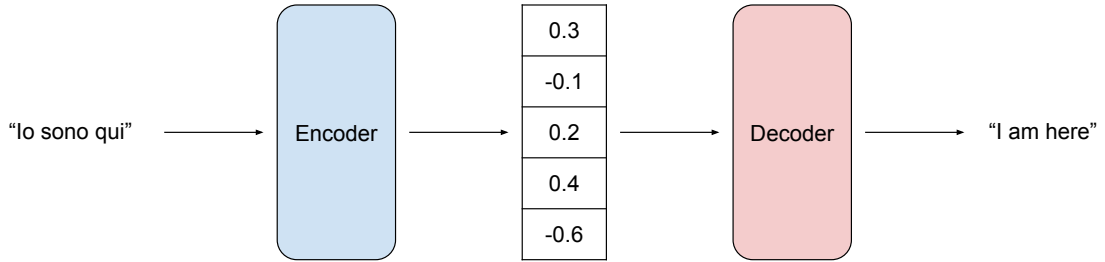


Figura 2: Schema semplificato di un'architettura encoder-decoder che traduce da italiano a inglese

L'idea che gli approcci in questa sezione propongono è quindi quella di trattare il problema della correzione degli errori come un problema di traduzione di sequenze: dato il dominio  $E$  delle sequenze contenenti errori, e il dominio  $C$  delle sequenze corrette (detto anche Ground Truth, abbreviato GT), il modello seq2seq che effettua la traduzione può essere visto come la seguente funzione:

$$M : E \rightarrow C \quad (2)$$

Allo stesso modo, ogni modello necessita prima di essere allenato con corpus parallelo dove ad ogni sequenza  $e \in E$  corrisponde una sequenza corretta  $c \in C$ .

**Modelli Word Based e Character based** Come si è visto, i modelli seq2seq trasformano una sequenza in un'altra sequenza. In base al tipo di elementi che compongono tali sequenze, è possibile descrivere due tipologie di modelli:

- **Modelli Word Based** (da qui in poi riferiti come WB): in questo tipo di modelli un elemento corrisponde ad una parola, o meglio, ad un token. Questo tipo di approcci richiede quindi che tutte le sequenze in  $E$  e in  $C$  siano tokenizzate. Inoltre, prima dell'allenamento, è necessario estrarre i dizionari della lingua d'origine e della lingua target dai corpora utilizzati.
- **Modelli Character Based** (da qui in poi riferiti come CB): in questo tipo di modelli un elemento corrisponde ad un carattere. Prima del training, è necessario estrarre il set di caratteri della lingua d'origine e della lingua target dai corpora utilizzati.

Da un'analisi della letteratura, si evince come i modelli CB siano generalmente preferiti per il task dell'OCR Post-Processing[14, 15, 16, 17, 18, 19]. In [14] viene proposto un confronto sullo stesso corpus fra due architetture molto simili, la prima WB e la seconda CB. Nello studio, gli autori concludono come l'architettura character based produca risultati significativamente migliori. I modelli CB infatti riescono a correggere gli errori a livello di carattere, e ciò comporta i seguenti vantaggi:

- A differenza dei modelli WB, i modelli CB possono correggere anche errori in parole non presenti nel dizionario estratto.
- Rispetto ai modelli WB, sono richiesti meno dati durante la fase di allenamento.

**Approcci basati su NMT** Il primo approccio presentato è quello descritto in [16]. Nella soluzione presentata gli autori utilizzano il tool OpenNMT[20] per allenare un modello CB. Il modello è allenato sia con coppie di sequenze identiche (quindi senza errori) che con sequenze dove la frase acquisita tramite OCR e la GT differiscono. Per queste ultime è adottata la seguente strategia di data augmentation: per ogni errore, si costruiscono cinque 5-gram con il token errato e i suoi token vicini facendo scorrere una finestra sopra la sezione di testo contenente il token errato. In questo modo si producono 5 sequenze diverse che possono essere usate per il training, evitando di avere un modello con troppo bias verso le sequenze senza errori. L'approccio per la correzione si divide dunque in due fasi: una prima fase di error detection utilizza un modello BERT (si rimanda il lettore alla sottosezione 1.1.3) messo a punto per la classificazione per individuare gli errori nelle sequenze; una seconda fase di error correction invece utilizza il modello di NMT allenato precedentemente per correggere gli errori.

In [19] si utilizza una metodologia simile per il training del modello, sviluppato con il framework Nematus[21]. Questo approccio utilizza una tecnica chiamata "factored NMT", utilizzata anche in [16], che permette di aggiungere informazioni strutturate alle sequenze in input. Ad esempio, è possibile aggiungere ad ogni carattere della sequenza informazioni sull'identificativo e l'anno del corpus di provenienza. Ciò permette di usare più corpora eterogenei per l'allenamento di un solo modello, aumentando così i dati a disposizione.

[17] propone invece un approccio per correggere unicamente i word segmentation errors (si rimanda il lettore alla Sezione 3.1), ovvero errori introdotti da spazi spuri che influiscono sulla segmentazione del testo. La metodologia proposta allena un modello su coppie di sequenze così composte: la sequenza target è una sequenza corretta senza errori; la sequenza di input invece è la sequenza target senza spaziature. In questo modo, il modello è allenato per inserire correttamente gli spazi

e può essere usato per correggere i word segmentation error.

In [15] si propone invece un approccio per il post processing e l'allenamento di un modello di NMT senza disporre di un corpus parallelo. L'approccio propone di allenare modello Word2Vec direttamente sul corpus contenente gli errori introdotti dal software di OCR. Dopodiché, per ogni parola estratta corretta (la correttezza è verificata attraverso un dizionario) si usa il modello Word2Vec per trovare le parole semanticamente più simili. Di queste, quelle con distanza di Levenshtein minore o uguale di 3 sono considerate versioni errate della parola corretta. Successivamente, le coppie di parole corrette e parole versione errate sono usate per allenare il modello di NMT, che è poi utilizzato per la correzione. Dato che il risultante modello è allenato per la correzione di singole parole, gli errori di segmentazione non vengono corretti. Inoltre, si rende necessaria una fase di error detection basata su un dizionario.

L'approccio proposto è ulteriormente sviluppato in [18]. Gli autori procedono sempre con la creazione di un corpus parallelo per l'allenamento di un modello di NMT, ma con una diversa strategia. In questo caso si parte da sequenze senza errori, che fungono da input per l'allenamento, e si creano sequenze target mediante l'introduzione di errori in maniera controllata. Inoltre, in questo caso, le sequenze usate per l'allenamento sono degli 5-gram centrati sulla parola errata, con lo scopo di fornire un maggior contesto per migliorare la correzione.

### 1.1.3 Approcci basati su BERT

**BERT** BERT[22] (Bidirectional Encoder Representation from Transformers) è un'architettura per la costruzione di modelli linguistici sviluppata da Google sulla base dei modelli transformers. Una delle differenze fra BERT e un transformer base consiste nella sola presenza di un encoder. Infatti, essendo BERT progettato solo per la modellazione linguistica (e non per la trasformazione di sequenze), non si rende necessaria la presenza di un decoder. Inoltre, a differenza dei modelli direzionali, in cui l'encoder legge il testo in input da sinistra verso destra o viceversa, l'encoder di BERT legge l'intera sequenza all'unisono. Ciò consente al modello di comprendere il contesto di una parola in base al testo a destra e sinistra di quest'ultima. I modelli BERT sono pre-allenati minimizzando la perdita combinata per i task di Masked Language Modelling (MLM) (sottosezione 3.2.2) e Next Sentence Prediction (NSP). Dopo la fase di pre-allenamento è possibile eseguire il fine-tuning del modello per uno specifico task di NLP, come Classification, Question Answering e Named Entity Recognition (NER).

**Approcci basati su BERT** In [16] gli autori eseguono il fine tuning di un modello BERT per eseguire error detection in un dataset in lingua inglese. Ciò

avviene dividendo in un primo momento la frase in input in sub-token tramite WordPiece. Il modello BERT classifica quindi i subtoken come validi o non validi: i token che contengono subtoken non validi sono segnalati quindi come errore.

In [23] gli autori usano una combinazione di BERT e FastText per generare possibili candidati per la correzione di errori introdotti da software OCR in testi in lingua inglese. Limitandosi ai 50 candidati più probabili prodotti con entrambi i metodi, la giusta correzione è presente fra i candidati nel 71% dei casi. Non è tuttavia proposto un metodo per scegliere il candidato corretto.

In [24] gli autori combinano un neural network di classificazione binaria per l'error detection e un'ulteriore neural network basato su BERT per l'error correction. I token (che in questo caso sono caratteri, dato che l'approccio lavora su testi in lingua cinese) marcati come errore sono mascherati dal correction network, che propone e sceglie successivamente le soluzioni con più probabilità di essere corrette. I due network sono allenati congiuntamente con un approccio end-to-end, con la loss function che è data da una combinazione lineare delle loss function di correction e detection network.

# Capitolo 2

## Dataset e Perturbazione

In questo capitolo è descritto il processo di creazione del dataset usato per valutare il sistema di correzione sviluppato.

Nella Sezione 2.1 sono discussi gli obiettivi e le caratteristiche che il dataset deve avere. Nella Sezione 2.2 sono delineate la metodologia di creazione e la struttura del dataset. Nella Sezione 2.3 è descritto il processo introduzione di errori all'interno delle frasi per simularne l'acquisizione tramite OCR, detto perturbazione. Nella Sezione 2.4 infine, sono discussi i parametri e la configurazione del dataset.

### 2.1 Introduzione

Lo sviluppo di un sistema di OCR post-processing non può prescindere dall'esecuzione di test per valutare, migliorare e confrontare le soluzioni adottate. Ciò rende necessario l'utilizzo di un dataset sul quale poter eseguire tutte le batterie di test necessarie. Tale dataset deve inoltre essere strutturato in modo tale da permettere di calcolare facilmente le metriche descritte nella Sezione 4.2. Se infatti lo scopo di un sistema di OCR post-processing è quello di correggere gli errori introdotti dai software di OCR, è necessario conoscere la posizione degli errori per controllare se siano stati corretti o meno.

La soluzione più semplice a questo problema è quella di disporre, per ogni frammento del dataset, sia del testo acquisito tramite OCR (e quindi contenente errori), sia del testo originale, detto anche ground truth. Non è però semplice trovare dataset di questo tipo, specialmente se in lingua italiana. Spesso, infatti, è necessario che la ground truth sia acquisita manualmente con un grosso dispendio di tempo e risorse: a causa di ciò, questo approccio non è stato considerato percorribile ai fini della tesi.

Si è invece deciso di adottare un approccio differente, che consiste nell'introduzione artificiale di errori all'interno di testo nativamente digitale (e quindi senza

errori) per simularne l'acquisizione tramite OCR. I vantaggi del processo appena descritto, detto "perturbazione", sono riassumibili nei seguenti punti:

- Produrre il dataset, una volta definita la funzione che esegue la perturbazione, è molto meno oneroso in termini di tempo e risorse. L'unico requisito è quello di procurarsi una collezione sufficientemente ampia di testo in formato digitale. Ciò è relativamente semplice, e può essere fatto, ad esempio, tramite web scraping.
- È possibile definire arbitrariamente l'intensità e la tipologia degli errori nel testo tramite l'uso di diverse funzioni di perturbazione. Ciò permette, dato un singolo testo di partenza, di ottenere testi con diverse intensità e tipologie di errori. In questo modo si rende possibile valutare le performance del sistema di OCR post-processing al variare delle condizioni del testo dato in input.

Il principale rischio che si corre utilizzando l'approccio appena descritto risiede nel fatto che il testo perturbato potrebbe non rispecchiare fedelmente gli errori presenti in testo acquisito realmente tramite software di OCR.



## 2.2 Creazione del dataset

Il dataset di partenza è una collezione di 15073 documenti in formato JSON, ottenuti mediante il web scraping del sito ufficiale del vaticano [www.vatican.va](http://www.vatican.va). I documenti contengono preghiere, lettere, discorsi, encicliche ecc. editi da figure ecclesiastiche dal 1439 al 2021.

Ogni documento è caratterizzato dalla struttura descritta in Tabella 1.

Nome campo	Contenuto
title	Titolo del documento
description	Insieme di keyword del documento
author	Autore del testo nel documento
creator	Coincide con author
language	Codice della lingua del documento (es. it, fr)
date	Data di redazione del documento
url	Url da cui è stato il documento
class	Tipologia di documento (es. discorso, preghiera...)
text	Testo completo estratto dal documento (titolo compreso)
url_references	Link al feed RSS del sito del vaticano
text_references	Posizione della pagina nella gerarchia del sito
italic	Testo estratto in corsivo nella pagina
paragraphs	Testo estratto diviso in paragrafi contenenti id e testo

Tabella 1: Struttura di un documento nel dataset

Dato il dataset descritto, l'obiettivo è quello di trasformarlo in un formato consono al testing. Nello specifico, è necessario frammentare il testo contenuto nei documenti in frasi di lunghezza minima e massima predeterminate. Le ragioni di questa decisione sono approfondite nel Capitolo 4, ma, in breve, spezzare il testo in frammenti più piccoli facilita il processo di allineamento delle frasi e quindi la valutazione delle correzioni. Per ognuno dei frammenti si vogliono poi produrre più versioni perturbate con diverse funzioni di perturbazione. Infine, è necessario che le frasi siano accompagnate da dei metadati che consentano la ricomposizione del testo originale mediante il riordinamento dei frammenti.

La creazione del dataset per il testing, descritta nelle prossime sezioni, segue dunque le seguenti fasi:

1. Filtraggio lingue
2. Estrazione
3. Frammentazione

4. Filtraggio paragrafi
5. Perturbazione
6. Riduzione

**Filtraggio lingue** In questa fase vengono filtrati tutti i documenti la cui lingua non è presente fra le lingue scelte. Dato l'insieme dei documenti  $D$  e l'insieme delle lingue consentite  $L$ , l'insieme filtrato dei documenti  $D_{fl}$  si ottiene come segue:

$$D_{fl} = \{d \mid \forall d \in D \mid language_d \in L\} \quad (3)$$

dove, dato un documento  $d \in D$ ,  $language_d$  è la lingua in cui è redatto. Le lingue scelte per la creazione del dataset sono elencate nella Sezione 2.4.

**Estrazione** In questa fase sono scartati i campi superflui ai fini del testing, e sono mantenuti solo i paragrafi assieme ad alcuni metadati utili a ricomporre il testo originale in seguito. È possibile formalizzare la prima parte di questa fase come segue:

$$D_{es1} = \{meta(paragraphs_d) \mid \forall d \in D\} \quad (4)$$

dove la funzione  $meta$  aggiunge ai paragrafi in  $paragraphs_d$  l'id del documento di provenienza. Si ottiene quindi un insieme di insiemi di paragrafi, dove ogni paragrafo è strutturato come in Tabella 2.

Nome campo	Contenuto
text	Testo contenuto nel paragrafo
parId	Codice del paragrafo all'interno del documento
docId	Codice del documento di provenienza del paragrafo.

Tabella 2: Struttura di un paragrafo

La fase di estrazione è completata appiattendo le liste di paragrafi nell'insieme  $D_{es1}$

$$D_{es} = \bigcup_{p \in D_{es1}} p \quad (5)$$

$D_{es}$  contiene quindi tutti i paragrafi del dataset iniziale strutturati come mostrato in Tabella 2.

**Frammentazione** Lo scopo di questa fase è quello di scomporre i paragrafi estratti nella fase precedente in frammenti più brevi, detti frasi, con le seguenti caratteristiche:

- Ogni frase è disgiunta dalle altre frasi estratte dal medesimo paragrafo.
- Ogni frase è compresa fra una lunghezza minima  $l_{min}$  e una lunghezza massima  $l_{max}$  (Sezione 2.4).

Dato quindi il testo di un paragrafo  $text_p$ , la frammentazione avviene secondo la seguente procedura:

1. Se la lunghezza di  $text_p$  è minore di  $l_{min}$ , non si procede e si scarta la stringa.
2. Si divide  $text_p$  su un segno di punteggiatura fra i seguenti: "?", "!", ";", ":", ".", ",", ". Se sono disponibili più opzioni (ovvero più punti in cui è possibile dividere su segni di punteggiatura) si divide nel punto minore o uguale a  $l_{max}$  che più si avvicina al valore di  $l_{max}$ . La parte a sinistra del punto di divisione è una frase estratta. La parte a destra invece viene riutilizzata come input della procedura di frammentazione, tornando al punto 1. Se non fosse possibile eseguire alcuna divisione, si passa al punto successivo.
3. Si divide  $text_p$  su uno spazio. Se sono disponibili più opzioni (ovvero più punti in cui è possibile dividere su uno spazio) si divide nel punto minore o uguale a  $l_{max}$  che più si avvicina al valore di  $l_{max}$ . La parte a sinistra del punto di divisione è una frase estratta. La parte a destra invece viene riutilizzata come input della procedura di frammentazione, tornando al punto 1. Se non fosse possibile eseguire alcuna divisione, si passa al punto successivo.
4. Se la lunghezza di  $text_p$  è minore di  $l_{max}$ ,  $text_p$  è aggiunto alle frasi estratte. Altrimenti, si divide  $text_p$  in posizione  $l_{max}$ . La parte a sinistra del punto di divisione è una frase estratta. La parte a destra invece viene riutilizzata come input della procedura di frammentazione, tornando al punto 1.

La procedura appena descritta fa in modo che i frammenti approssimino  $l_{max}$ , evitando il più possibile introdurre errori. Spezzare un paragrafo nel mezzo di una parola, ad esempio, andrebbe a creare due frammenti contenenti degli errori (uno sulla parola finale del primo, l'altro sulla parola iniziale del secondo).

Durante la frammentazione ogni frammento ritiene i metadati relativi al documento e al paragrafo di appartenenza. In più, si aggiunge un ulteriore campo per tenere traccia della posizione del frammento all'interno del paragrafo di appartenenza. In questo modo, è possibile ricomporre un documento in un secondo momento. Ogni frase prodotta rispetta il seguente schema:

Nome campo	Contenuto
text	Testo contenuto nella frase
parId	Codice del paragrafo di provenienza
docId	Codice del documento di provenienza.
parPos	Posizione della frase all'interno del paragrafo di provenienza

Tabella 3: Struttura di una frase

La fase di frammentazione si può quindi formalizzare in due step. Nel primo step, ogni paragrafo viene frammentato nella lista delle sue frasi:

$$D_{fr1} = \{extr(p) \mid p \in D_{es}\} \quad (6)$$

dove *extr* è la funzione che estrae le frasi secondo la procedura descritta precedentemente, producendo un insieme di frasi strutturate come mostrato in Tabella 3. Nel secondo step si appiattisce l'insieme di insiemi che si è andato a creare, per ottenere insieme di frasi:

$$D_{fr} = \bigcup_{f \in D_{fr1}} f \quad (7)$$

**Filtraggio paragrafi** Lo scopo di questa fase è quello di rimuovere dal dataset tutte le frasi che appartengono al primo o all'ultimo paragrafo di un documento. Ciò si rende necessario perchè questi paragrafi spesso contengono intestazioni o piè di pagina ripetitivi che andrebbero a sporcare il dataset. Ad esempio, l'ultimo paragrafo contiene spesso una frase simile a "*© Copyright 2004 - Libreria Editrice Vaticana*", mentre il primo spesso contiene solamente una data. È definita la seguente funzione *maxPar* che, data una frase  $f \in D_{fr}$ , indica se  $f$  appartiene all'ultimo paragrafo del documento da cui è stata estratta:

$$maxPar(f) = \begin{cases} 1 & \text{se } f \text{ appartiene all'ultimo paragrafo di } docId_f \\ 0 & \text{altrimenti} \end{cases} \quad (8)$$

L'operazione di filtraggio dei paragrafi può quindi essere formalizzata come segue:

$$D_{fp} = \{f \mid f \in D_{fr} \mid parId_f \neq 0 \wedge maxPar(f) = 0\} \quad (9)$$

**Perturbazione** Durante la fase di perturbazione, per ogni frase del dataset sono generate le sue versioni perturbate. Ogni versione perturbata di una frase viene generata da una diversa funzione di perturbazione applicata alla frase stessa. Il funzionamento delle funzioni di perturbazione è spiegato nella Sezione 2.3, mentre le esatte funzioni per la creazione del dataset sono elencate nella Sezione 2.4.

Viene chiamata  $F_p$  la lista  $[p_1, \dots, p_n]$  delle funzioni di perturbazione utilizzate, in cui ogni  $p_i$  è identificata da un codice. Data una stringa di testo  $s$ , la sua versione perturbata dalla funzione  $p_i$  corrisponde a  $p_i(s)$ .

Data quindi una frase  $f \in D_{fr}$  strutturata come in Tabella 3, e lista delle funzioni di perturbazione  $F_p$ , si definisce come  $P_{list}$  l'insieme delle frasi perturbate  $[p_1(text_f), \dots, p_n(text_f)]$  la cui rappresentazione è schematizzata in Tabella 4.

Codice funzione perturbazione	Frase perturbata
<i>nome_func</i> <sub>1</sub>	$p_1(f)$
<i>nome_func</i> <sub>2</sub>	$p_2(f)$
...	...
<i>nome_func</i> <sub>n</sub>	$p_n(f)$

Tabella 4: Struttura delle frasi perturbate

Si definisce inoltre la funzione *applyPert* che, data una frase  $f \in D_{fr}$ , produce un sample aggiungendo a  $f$  un campo con le sue versioni perturbate della frase originale. Un sample è strutturato come in Tabella 5.

Nome campo	Contenuto
text	Testo contenuto nella frase non perturbata
parId	Codice del paragrafo di provenienza
docId	Codice del documento di provenienza.
parPos	Posizione della frase all'interno del paragrafo di provenienza
perturbed	Frase perturbate ( $[p_1(text_f), \dots, p_n(text_f)]$ )

Tabella 5: Struttura di una frase

La fase di perturbazione può quindi essere formalizzata come segue:

$$D_{pe} = \{applyPert(f) \mid f \in D_{fr}\} \quad (10)$$

**Riduzione** Lo scopo della fase di riduzione è quello di creare una versione ridotta del dataset, ovvero quello di ridurre il numero di elementi. Ciò è reso necessario dal fatto che il testing può essere significativamente oneroso in termini di tempo e risorse, specialmente su dataset troppo ampi.

Data quindi la dimensione del dataset ridotto  $n$  (i cui valori sono specificati nella Sezione 2.4), il dataset  $D_{re}$  è un sottoinsieme di  $n$  elementi estratti casualmente da  $D_{pe}$ :

$$D_{re} \subset D_{pe} \wedge |D_{re}| = n \quad (11)$$

Dopo aver eseguito la fase di riduzione, la costruzione del dataset può dirsi completata.

## 2.3 Perturbazione

Data una stringa  $s$ , lo scopo del processo di perturbazione è quello produrre una stringa  $s'$  nella quale sono introdotti degli errori per simulare l'acquisizione del testo tramite OCR. Per permettere l'introduzione controllata e a diverse intensità degli errori è stato sviluppato un sistema modulare e componibile, che sarà illustrato in questa sezione.

### 2.3.1 Moduli di perturbazione

La componente base del sistema di perturbazione è il modulo di perturbazione. Un modulo di perturbazione non è altro che una funzione che modella l'inserimento di una specifica tipologia di errore. Ciò significa che per ogni tipologia di errore modellata si rende necessaria la creazione di un modulo di perturbazione. Ad esempio, è possibile definire un modulo di perturbazione che modelli la sostituzione di alcuni caratteri, o un modulo che inserisca della punteggiatura spuria. Tutti i moduli di perturbazione, data una lista ordinata di token, introducono errori mediante la modifica, la divisione, l'unione o la cancellazione di uno o più token.

Data una lista di token  $list_t = [t_1, \dots, t_n]$ , Il funzionamento di un modulo di perturbazione si può dividere nelle seguenti tre fasi:

1. **Raggruppamento:** I token sono raggruppati in gruppi consecutivi e disgiunti di  $k$  elementi, dove  $k$  varia a seconda della tipologia di modulo di perturbazione.

La fase di raggruppamento è quindi definita come segue:

$$group : [t_1, \dots, t_n] \rightarrow [g_1, \dots, g_q] \quad (12)$$

dove ogni  $g_i$  è un gruppo di token  $[t_1, \dots, t_k]$  e  $q = \lceil n/k \rceil$ .

Ad esempio, data la lista di token  $['Nel', 'mezzo', 'del', 'cammin', 'di', 'nostra', 'vita']$  e  $k = 2$ , i token sono raggruppati nel seguente modo:  $['Nel', 'mezzo'], ['del', 'cammin'], ['di', 'nostra'], ['vita']$ .

2. **Perturbazione:** Ogni tipologia di modulo di perturbazione è caratterizzata da una specifica funzione di perturbazione  $f_{pert}$

$$f_{pert} : [t_1, \dots, t_k] \rightarrow [t'_1, \dots, t'_j] \quad (13)$$

che introduce errori mediante la modifica, la divisione, l'unione o la cancellazione dei token in un determinato gruppo, con una probabilità  $p$  definita per ogni modulo. Un qualsiasi gruppo di token  $g \in [g_1, \dots, g_q]$  viene perturbato o meno secondo la seguente funzione:

$$f_{map}(g) = \begin{cases} f_{pert}(g) & \text{con probabilità } p \\ g & \text{con probabilità } 1 - p \end{cases} \quad (14)$$

È quindi possibile definire la fase di perturbazione come:

$$pert : [g_1, \dots, g_q] \rightarrow [g'_1, \dots, g'_q] \quad (15)$$

dove  $g'_i = f_{map}(g_i) \forall i \in [1..q]$ .

Ad esempio, si supponga di avere un modulo in cui  $f_{pert}$  sia definita come:

$$f_{ex}([t_1, t_2]) = [t_1 \frown t_2] \quad (16)$$

La funzione appena definita concatena i due token all'interno di gruppo, restituendo un gruppo formato da un solo token. Si consideri l'esempio nel punto precedente, e si supponga che l'unico gruppo affetto da perturbazione sia il secondo. Si ha quindi che  $f_{pert}(['del', 'cammin']) = ['delcammin']$ , e quindi alla fine di questa fase si ottiene:  $['Nel', 'mezzo'], ['delcammin'], ['di', 'nostra'], ['vita']$ .

3. **Appiattimento:** la fase di appiattimento è l'opposto di quanto avviene durante il raggruppamento. I gruppi di token sono raggruppati nuovamente in un'unica lista:

$$flat : [g'_1, \dots, g'_q] \rightarrow [t_1, \dots, t_m] \quad (17)$$

dove  $m$  potrebbe coincidere o meno con  $n$  a seconda del tipo di perturbazione avvenuta.

Riprendendo l'esempio precedente, il risultato di questa fase è:  $['Nel', 'mezzo', 'delcammin', 'di', 'nostra', 'vita']$

Come si può notare, un modulo riceve in input una lista di token e fornisce in output un'altra lista di token. Ciò rende possibile concatenare più moduli, per introdurre più di un tipo di errore all'interno di una sequenza di token. Questa

eventualità è descritta nella sottosezione 2.3.3 e nella sottosezione 2.3.4.

Quanto descritto finora non basta a soddisfare lo scopo della perturbazione, che è quello di trasformare una stringa in una nuova stringa contenente degli errori. Tale funzione è resa possibile dai moduli speciali, che sono approfonditi nella sottosezione 2.3.2.

Dal funzionamento dei moduli di perturbazione si evince come la funzione  $f_{pert}$  caratterizzi il modulo modellando il tipo di errore che esso va ad introdurre. Sono quindi stati creati i seguenti moduli, ognuno dei quali è caratterizzato da una diversa funzione di perturbazione:

- Modulo di space splitting ( $M_{ss}$ )
- Modulo di punctuation splitting ( $M_{ps}$ )
- Modulo di punctuation insertion ( $M_{pi}$ )
- Modulo di hyphen merging ( $M_{hm}$ )
- Modulo di characters replacement ( $M_{cr}$ )

Ognuno di questi moduli è stato creato per modellare un particolare tipo di errore. Le tipologie di errore modellate sono state decise in base all'analisi di documenti il cui testo è stato acquisito tramite OCR. Più nello specifico, si tratta delle trascrizioni dei dibattiti parlamentari della prima e della seconda legislatura della repubblica italiana.

### Modulo di space splitting

Il modulo di space splitting modella l'errore in cui le lettere di una parola sono intramezzate da degli spazi. Siccome la perturbazione avviene su un singolo token, i token in input sono raggruppati in gruppi di un token ( $k = 1$ ). Si definisce dunque la funzione di perturbazione del modulo:

$$f_{pert\_split}([t_1]) = [f_{split}(t_1)] \quad (18)$$

in cui  $f_{split}$  è la funzione che aggiunge uno spazio dopo ogni carattere di un token.

### Esempi

$$f_{pert\_split}(["cammin"]) = ["c a m m i n "] \quad (19)$$

$$f_{pert\_split}(["nostra"]) = ["n o s t r a "] \quad (20)$$



### Modulo di punctuation splitting

Il modulo di punctuation splitting modella l'errore in cui una o più occorrenze di un segno di punteggiatura intramezzano i caratteri di una parola. Siccome la perturbazione avviene su un singolo token, i token in input sono raggruppati in gruppi di un token ( $k = 1$ ). Si definisce dunque la funzione di perturbazione del modulo:

$$f_{pert\_punctsplit}([t_1]) = [f_{punctsplit}(t_1, punct)] \quad (21)$$

Assumendo che:

- *punct* sia il segno di punteggiatura che è intramezzato agli spazi,
- un token  $t$  sia una sequenza di caratteri  $[c_1, \dots, c_n]$ , ovvero  $|t| = n$ ,
- sia definito un numero casuale  $x \in \mathbb{N} \mid 1 \leq x \leq n$

è possibile definire  $f_{punctsplit}$  come:

$$f_{punctsplit}(t, punct) = \begin{cases} [c_1, \dots, c_x] \frown punct \frown f_{punctsplit}([c_{x+1}, \dots, c_n], punct) & \text{se } x < |t| \\ t & \text{se } x \geq |t| \end{cases} \quad (22)$$

Informalmente, la funzione aggiunge un carattere *punct* ogni  $x$  caratteri all'interno di  $t$ .

### Esempi

$$f_{punctsplit}("cammin", ",") = "ca,mm,in" \quad (x = 2) \quad (23)$$

$$f_{punctsplit}("mezzo", ",") = "m,e,z,z,o" \quad (x = 1) \quad (24)$$

### Modulo di punctuation insertion

Il modulo di punctuation insertion modella l'errore in cui un segno di punteggiatura è aggiunto fra due token, senza quindi dividere in due un singolo token. Siccome la perturbazione avviene su un singolo token, i token in input sono raggruppati in gruppi di un token ( $k = 1$ ). Si definisce dunque la funzione di perturbazione del modulo:

$$f_{pert\_insertion}([t_1]) = [t_1, punct] \quad (25)$$

dove *punct* è il segno di punteggiatura da aggiungere.

**Esempi** Data la sequenza di gruppi di token  $['Nel']$ ,  $['mezzo']$ ,  $['del']$ ,  $['cammin']$ ,  $['di']$ ,  $['nostra']$ ,  $['vita']$ , dove i gruppi sottolineati sono quelli da perturbare, e dato  $punct = "."$ , si ha che:

$$f_{pert.insertion}(["Nel", "."]) = ["Nel", "."] \quad (26)$$

$$f_{pert.insertion}(["cammin", "."]) = ["cammin", "."] \quad (27)$$

Quindi, applicata la fase di appiattimento, la sequenza finale risulta:  $['Nel']$ ,  $['.']$ ,  $['mezzo']$ ,  $['del']$ ,  $['cammin']$ ,  $['.']$ ,  $['di']$ ,  $['nostra']$ ,  $['vita']$ .

### Modulo di hyphen merging

Il modulo di hyphen merging modella l'errore in cui due parole consecutive vengono unite da un trattino. Dovendo unire due token tra di loro, la perturbazione avviene in gruppi di due token, quindi si ha  $k = 2$ . Si definisce dunque la funzione di perturbazione del modulo:

$$f_{hyphen}([t_1, t_2]) = [t_1 \frown "-" \frown t_2] \quad (28)$$

**Esempio** Data la sequenza di gruppi di 2 token  $['Nel', 'mezzo']$ ,  $['del', 'cammin']$ ,  $['di', 'nostra']$ ,  $['vita']$ , dove il gruppo sottolineato è quello da perturbare, si ha che:

$$f_{hyphen}(["di", "nostra"]) = ["di-nostra"] \quad (29)$$

Quindi, applicata la fase di appiattimento, la sequenza finale risulta:  $['Nel']$ ,  $['mezzo']$ ,  $['del']$ ,  $['cammin']$ ,  $['di-nostra']$ ,  $['vita']$ .

### Modulo di characters replacement

Il modulo di characters replacement modella l'errore in un cui avviene la cancellazione, l'aggiunta o la sostituzione di uno o più caratteri all'interno di un token. Siccome la perturbazione avviene su un singolo token, i token in input sono raggruppati in gruppi di un token ( $k = 1$ ). Si definisce dunque la funzione di perturbazione del modulo:

$$f_{charrepl}([t_1]) = [f_{alternative}(t_1)] \quad (30)$$

Per ogni token  $t$  che viene perturbato, esiste un insieme di 5 versioni errate di  $t$ , detto  $A_t = [a_{t1}, \dots, a_{t5}]$ , generato sulla base di una distribuzione di errori preesistente. La funzione  $f_{alternative}$  rimpiazza  $t$  con una delle alternative in  $A_t$ :

$$f_{alternative}(t) = a_{ti} \quad (31)$$

dove  $a_{ti} \in A_t$  e  $i$  è un numero intero casuale fra 1 e 5.

**Esempio** Sia dato il token "cammin" e si supponga l'insieme delle versioni errate del token essere  $A_{cammin} = ['cmmmin', 'camniin', 'cdmmn', 'camin', 'cammmii']$ . Si sceglie quindi in modo casuale un token dall'insieme  $A_{cammin}$ :

$$f_{alternative}("cammin") = a_{cammin.2} = "camniin" \quad (32)$$

### 2.3.2 Moduli speciali

Come si è visto nella sottosezione precedente, i moduli di perturbazione lavorano su sequenze di token, nelle quali introducono degli errori. È però necessario, come accennato all'inizio della Sezione 2.3, che la perturbazione prenda in input e restituisca come output delle stringhe. Sono quindi definiti due moduli, detti moduli speciali, che hanno lo scopo di convertire una stringa in una sequenza di token e viceversa:

- Modulo di tokenizzazione ( $M_{to}$ )
- Modulo di detokenizzazione ( $M_{de}$ )

#### Modulo di tokenizzazione

Il modulo di tokenizzazione ha lo scopo di tokenizzare una qualsiasi stringa  $s \in S$  in una lista  $[t_1, \dots, t_n]$  di token:

$$tok : S \rightarrow [t_1, \dots, t_n] \quad (33)$$

**Esempio** Data la frase  $s_{ex} = "Nel mezzo del cammin di nostra vita"$

$$tok(s_{ex}) = ['Nel', 'mezzo', 'del', 'cammin', 'di', 'nostra', 'vita'] \quad (34)$$

#### Modulo di detokenizzazione

Il modulo di detokenizzazione ha lo scopo di detokenizzare, ovvero di eseguire il processo contrario alla tokenizzazione, una lista di token  $[t_1, \dots, t_n]$  in una stringa  $s \in S$ .

$$detok : [t_1, \dots, t_n] \rightarrow S \quad (35)$$

**Esempio** Data la sequenza di token  $T_{ex} = ['Nel', 'mezzo', 'del', 'cammin', 'di', 'nostra', 'vita']$ .

$$detok(T_{ex}) = "Nel mezzo del cammin di nostra vita" \quad (36)$$

### 2.3.3 Pipeline

Come accennato nelle sottosezioni precedenti, ogni modulo di perturbazione modella un diverso tipo di errore. In realtà, in un testo estratto tramite OCR sono presenti più tipologie di errori. È quindi necessario utilizzare più moduli di perturbazione in sequenza per ottenere una perturbazione verosimile. Considerando che ogni modulo di perturbazione è considerabile come una funzione con dominio e codominio coincidenti  $T$  (ogni elemento del dominio e codominio  $T$  è una lista di token  $[t_1, \dots, t_n]$  con  $n \in \mathbb{N} \wedge n \geq 0$ ), è possibile concatenare più moduli insieme tramite composizione:

$$m_k \circ m_{k-1} \circ m_{k-2} \circ \dots \circ m_1 \quad (37)$$

dove ogni  $m_i$  è istanza di un modulo di perturbazione. La funzione composta non può però trattare stringe, ma solo sequenze di token. Ciò è ovviabile attraverso l'aggiunta dei moduli speciali di tokenizzazione e detokenizzazione all'inizio e alla fine della composizione. La funzione risultante prende il nome di pipeline:

$$Pipeline : detok \circ m_k \circ m_{k-1} \circ m_{k-2} \circ \dots \circ m_1 \circ tok \quad (38)$$

Data quindi una stringa  $s \in S$ , la stringa perturbata  $s' \in S$  è ottenuta come:

$$Pipeline(s) = s' \quad (39)$$

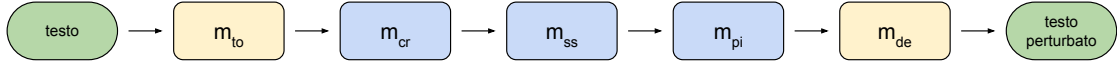
**Esempio** Si definisca una pipeline  $p_{ex}$  composta come indicato in Tabella 6.

Posizione	Nome istanza	Tipo Modulo	p	punct
1	$m_{to}$	$M_{to}$	n.d	n.d
2	$m_{cr}$	$M_{cr}$	0.3	n.d
3	$m_{ss}$	$M_{ss}$	0.1	n.d
4	$m_{pi}$	$M_{pi}$	0.1	,
5	$m_{de}$	$M_{de}$	n.d	n.d

Tabella 6: Struttura della pipeline  $p_{ex}$

$p_{ex}$  corrisponde dunque alla seguente funzione, schematizzata anche in Figura 3.

$$p_{ex} = m_{de} \circ m_{pi} \circ m_{ss} \circ m_{cr} \circ m_{to} \quad (40)$$

Figura 3: Schema della pipeline  $p_{ex}$ 

Data una stringa di esempio  $s_{ex} = \text{"Nel mezzo del cammin di nostra vita"}$ , è mostrata in seguito la sequenza dei passaggi che portano alla perturbazione della frase:

0.  $(s_{ex})$   
*"Nel mezzo del cammin di nostra vita"*
1.  $m_{to}(s_{ex})$   
 $['Nel', 'mezzo', 'del', 'cammin', 'di', 'nostra', 'vita']$
2.  $m_{cr}(m_{to}(s_{ex}))$   
 $['Nel', 'inezzo', 'del', 'camniin', 'di', 'nostra', 'vita']$
3.  $m_{ss}(m_{cr}(m_{to}(s_{ex})))$   
 $['Nel', 'inezzo', 'del', 'c a m n i i n', 'di', 'nostra', 'vita']$
4.  $m_{pi}(m_{ss}(m_{cr}(m_{to}(s_{ex}))))$   
 $['Nel', 'inezzo', 'del', 'c a m n i i n', 'di', ' ', 'nostra', 'vita']$
5.  $m_{de}(m_{pi}(m_{ss}(m_{cr}(m_{to}(s_{ex}))))))$   
*"Nel inezzo del c a m n i i n di, nostra vita"*

Si sottolinea come questa sia solo una delle possibili perturbazioni di  $s_{ex}$ : essendo nella perturbazione una componente aleatoria, ogni esecuzione di una pipeline su uno stesso input può dare risultati diversi.

### 2.3.4 Superpipeline

Una pipeline di perturbazione perturba il testo in modo uniforme: ogni frase perturbata con una pipeline  $p$  ha una concentrazione di errori simile. Nell'osservazione di testi acquisiti tramite OCR si nota però come la distribuzione degli errori non sia uniforme: si possono incontrare sezioni prive di errori, ma anche parti di testo in cui è presente molto rumore. Ciò è spesso dovuto alle condizioni del documento originale, che in alcune parti può essere più degradato che in altre. Questo comporta che una sola pipeline non può simulare fedelmente gli errori OCR: è necessario variare l'intensità e la modalità di perturbazione per ottenere un risultato più verosimile.

Si introduce quindi il concetto di superpipeline. Date:

- Una lista  $Ppl = [p_1, \dots, p_n]$  dove ogni  $p_i$  è una pipeline,
- Una lista di pesi  $W = [w_1, \dots, w_n]$  dove  $w_i \in \mathbb{N}$  è il peso associato ad  $p_i$ ,

è possibile definire una superpipeline come una funzione che, data una stringa  $s \in S$ , ne produce una versione perturbata tramite una pipeline  $p_{rand}$  scelta casualmente da  $Ppl$ . La probabilità che una pipeline  $p_i$  venga scelta corrisponde a:

$$P(p_{rand} = p_i) = \frac{w_i}{\sum_{j=0}^n w_j} \quad (41)$$

**Esempio** Si definiscono le tre pipeline  $p_{ex1}$ ,  $p_{ex2}$ ,  $p_{ex3}$ , rispettivamente in Tabella 7, Tabella 8, Tabella 9.

Posizione	Nome istanza	Tipo Modulo	p	punct
1	$m_{to}$	$M_{to}$	n.d	n.d
2	$m_{cr}$	$M_{cr}$	0.5	n.d
3	$m_{de}$	$M_{de}$	n.d	n.d

Tabella 7: Definizione di  $p_{ex1}$

Posizione	Nome istanza	Tipo Modulo	p	punct
1	$m_{to}$	$M_{to}$	n.d	n.d
2	$m_{cr}$	$M_{cr}$	0.3	n.d
3	$m_{hm}$	$M_{hm}$	0.1	n.d
4	$m_{de}$	$M_{de}$	n.d	n.d

Tabella 8: Definizione di  $p_{ex2}$

Posizione	Nome istanza	Tipo Modulo	p	punct
1	$m_{to}$	$M_{to}$	n.d	n.d
2	$m_{cr}$	$M_{cr}$	0.3	n.d
3	$m_{ss}$	$M_{ss}$	0.1	n.d
4	$m_{pi}$	$M_{pi}$	0.1	,
5	$m_{de}$	$M_{de}$	n.d	n.d

Tabella 9: Definizione di  $p_{ex3}$

Si definisce inoltre la lista dei pesi  $W = [1, 3, 2]$ . Quindi, calcolata la somma dei pesi  $w_1 + w_2 + w_3 = 6$ , si ha che:

- $p_{ex1}$  è usata con probabilità  $w_1/6 = 1/6 = 0.17$
- $p_{ex2}$  è usata con probabilità  $w_2/6 = 3/6 = 0.50$
- $p_{ex3}$  è usata con probabilità  $w_3/6 = 2/6 = 0.33$

La superpipeline composta è quindi schematizzabile come mostrato in Figura 4.

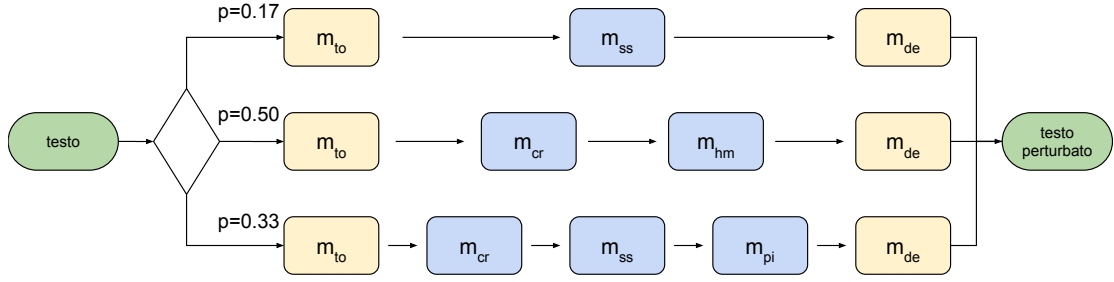


Figura 4: Schema della superpipeline descritta nell'esempio

## 2.4 Configurazione

Nelle precedenti sezioni sono stati descritti la metodologia di creazione del dataset e il funzionamento del sistema di perturbazione. In questa sezione sono invece descritti i parametri usati per il sistema di perturbazione e quindi per la creazione del dataset.

### 2.4.1 Pipeline e Superpipeline

In questa sottosezione sono definite le superpipeline usate per la perturbazione del dataset e le pipeline che le compongono. Sono state stabilite 3 tipologie di superpipeline, ognuna delle quali ha 3 livelli di intensità:

- **Token superpipeline:** sono superpipeline che introducono solo dei word error (Sezione 3.1), ovvero errori contenuti all'interno di un singolo token che non impattano la tokenizzazione. Le token superpipeline sono identificate dai codici T1, T2, T3, dove T1 e T3 sono rispettivamente la superpipeline che introduce meno errori e quella che ne introduce di più.
- **Segmentation superpipeline:** sono superpipeline che introducono solo dei word segmentation error (Sezione 3.1), ovvero che dividono o uniscono uno o più token e che impattano quindi la tokenizzazione. Le segmentation superpipeline sono identificate dai codici S1, S2, S3, dove S1 e S3 sono

rispettivamente la superpipeline che introduce meno errori e quella che ne introduce di più.

- **Mixed Pipeline:** sono superpipeline che introducono sia word error che word segmentation error. Le segmentation superpipeline sono identificate dai codici M1, M2, M3, dove M1 e M3 sono rispettivamente la superpipeline che introduce meno errori e quella che ne introduce di più.

In totale sono definite 9 superpipeline (T1, T2, T3, S1, S2, S3, M1, M2, M3): questo numero permette di avere una buona granularità tenendo comunque contenuti i tempi durante la fase di testing.

## Pipeline

Le superpipeline appena descritte sono formate dalla combinazione di più pipeline. Sono definiti tre tipi di pipeline:

- Token pipeline
- Segmentation pipeline
- Mixed pipeline

ognuna di queste tipologie di pipeline introduce gli stessi tipi di errore delle omonime superpipeline. Per ogni tipologia di pipeline sono definite 3 pipeline, ognuna delle quali ha gli stessi moduli con probabilità diverse.

**Token pipeline** Una token pipeline è formata dai moduli presenti nella seguente Tabella 10:

Posizione	Tipo Modulo	punct
1	$M_{to}$	n.d
2	$M_{cr}$	n.d
3	$M_{de}$	n.d

Tabella 10: Moduli presenti in una token pipeline

In Tabella 11 sono definite le tre token pipeline. Nella tabella,  $p_i$  indica la probabilità associata al modulo in posizione  $i$ .



<b>Codice</b>	<b><math>p_1</math></b>	<b><math>p_2</math></b>	<b><math>p_3</math></b>
$tok_1$	n.d	0.1	n.d
$tok_2$	n.d	0.3	n.d
$tok_3$	n.d	0.3	n.d

Tabella 11: Definizione delle tre token pipeline

**Segmentation pipeline** Una segmentation pipeline è formata dai moduli presenti nella seguente Tabella 12:

<b>Posizione</b>	<b>Tipo Modulo</b>	<b>punct</b>
1	$M_{to}$	n.d
2	$M_{hm}$	n.d
3	$M_{ps}$	, (virgola)
4	$M_{ss}$	n.d
5	$M_{pi}$	. (punto)
6	$M_{pi}$	, (virgola)
7	$M_{pi}$	' (apostrofo)
8	$M_{de}$	n.d

Tabella 12: Moduli presenti in una segmentation pipeline

In Tabella 13 sono definite le tre segmentation pipeline. Nella tabella,  $p_i$  indica la probabilità associata al modulo in posizione  $i$ .

<b>Codice</b>	<b><math>p_1</math></b>	<b><math>p_2</math></b>	<b><math>p_3</math></b>	<b><math>p_4</math></b>	<b><math>p_5</math></b>	<b><math>p_6</math></b>	<b><math>p_7</math></b>	<b><math>p_8</math></b>
$seg_1$	n.d	0.001	0.001	0.0025	0.005	0.005	0.005	n.d
$seg_2$	n.d	0.001	0.002	0.008	0.025	0.025	0.025	n.d
$seg_3$	n.d	0.01	0.02	0.05	0.1	0.1	0.1	n.d

Tabella 13: Definizione delle tre segmentation pipeline

**Mixed pipeline** Le mixed pipeline derivano dalla composizione di una token pipeline con una segmentation pipeline. Le mixed pipeline sono definite in Tabella 14.

<b>Codice</b>	<b>Definizione</b>
$mix_1$	$seg_1 \circ tok_1$
$mix_2$	$seg_2 \circ tok_2$
$mix_3$	$seg_3 \circ tok_3$

Tabella 14: Definizione delle tre mixed pipeline

## Superpipeline

Date le pipeline appena definite, le superpipeline si compongono come mostrato nella Tabella 15: ad ogni superpipeline sono associati i pesi di ciascuna delle sue pipeline.

Codice	$tok_1$	$tok_2$	$tok_3$	$seg_1$	$seg_2$	$seg_3$	$mix_1$	$mix_2$	$mix_3$
T1	6	4	1	/	/	/	/	/	/
T2	2	8	1	/	/	/	/	/	/
T3	1	6	4	/	/	/	/	/	/
S1	/	/	/	6	4	1	/	/	/
S2	/	/	/	2	8	1	/	/	/
S3	/	/	/	1	6	4	/	/	/
M1	/	/	/	/	/	/	6	4	1
M2	/	/	/	/	/	/	2	8	1
M3	/	/	/	/	/	/	1	6	4

Tabella 15: Definizione delle nove superpipeline

## 2.4.2 Dataset

In questa sottosezione sono descritte le istanze del dataset create e i loro parametri. Sono ricordati in seguito i parametri configurabili per la creazione di un dataset:

- $l_{min}$ : lunghezza minima di una frase nel dataset.
- $l_{max}$ : lunghezza massima di una frase nel dataset.
- *Lingue*: lista di lingue consentite nelle frasi nel dataset.
- *Superpipeline*: lista di superpipelines usate per la perturbazione.
- *Dimensione*: numero di elementi che formano il dataset, se esso viene ridotto

In Tabella 16 sono elencate le versioni del dataset definiti e i loro parametri:

Codice	$l_{min}$	$l_{max}$	Lingue	Superpipeline	Dimensione
dst@50	8	50	[it]	[T1, T2, T3, S1, S2, S3, M1, M2, M3]	10000
dst@100	20	100	[it]	[T1, T2, T3, S1, S2, S3, M1, M2, M3]	10000

Tabella 16: Configurazione delle versioni del dataset

Le due versioni del dataset differiscono solamente nella lunghezza massima e minima delle frasi. In questo modo sarà possibile testare quanto la lunghezza di

una frase (e quindi il contesto intorno ad un errore) influisca sulle performance di correzione. Questi aspetti, insieme ai risultati dei test, sono approfonditi nel Capitolo 4.



# Capitolo 3

## Metodologia di correzione

In questo capitolo è descritta la metodologia di correzione.

Nella Sezione 3.1 vengono descritti gli obiettivi del processo di correzione e le criticità che lo contraddistinguono. Nella sottosezione 3.2.1 è presente una panoramica generale del processo di correzione. Sono inoltre descritte le fasi e le componenti del sistema. Nella sottosezione 3.2.2 è descritto il funzionamento del BERT Masked Language Model. Nelle sottosezioni 3.2.3 e 3.2.4 è descritto il funzionamento dei moduli che stanno alla base del sistema di correzione.

### 3.1 Introduzione

L'Optical Character Recognition, o OCR, è una tecnologia tramite la quale è possibile estrarre e digitalizzare il testo presente in un'immagine. Più precisamente, "digitalizzare" significa tradurre il testo dell'immagine in una codifica leggibile da una macchina, come ad esempio ASCII o Unicode.

Provvedimenti per incrementare l'occupazione operaia, agevolando la costruzione di case per i lavoratori.

Prima di dare la parola al primo iscritto, mi permetto di far presente alla Camera che sono già stati presentati vari ordini del giorno e vari emendamenti, relativi a questo disegno di legge. Come la Camera sa, gli emendamenti, di regola, debbono esser presentati almeno ventiquattro ore prima della seduta in cui vengono discussi. Per eccezione, possono anche esser presentati senza rispettare questo termine, cioè nella seduta stessa, purché siano



Prima, di dare la parola al primo iscritto, mi permetto di far presente alla Camera che sono già stati presentati vari ordini del giorno e vari emendamenti, relativi a questo disegno di legge. Come la Camera sa, gli emendamenti, di regola, debbono esser presentati almeno ventiquattro ore prima della seduta in cui vengono discussi.

Figura 5: A sinistra un frammento di immagine contenente testo, a destra il testo digitalizzato estratto dal frammento di immagine.

L'uso di sistemi OCR è particolarmente vantaggioso per l'archiviazione dei documenti. Infatti, l'estrazione del testo rende possibile eseguire sui documenti operazioni di ricerca e di reperimento informazioni. Si pensi a un'operazione basilare come la ricerca di tutti i documenti che contengono una determinata parola: un task dispendioso e manuale che è quasi istantaneo se si ha a disposizione il testo digitalizzato.

**Errori** I sistemi OCR presentano però alcuni problemi che possono influenzare negativamente le performance nel reperimento di informazioni [3][4]. Si veda, ad esempio, il testo estratto in Figura 5: seppur la maggior parte delle parole sono state riconosciute correttamente, è possibile notare alcuni errori. Generalmente la presenza di tali errori è dovuta a piccole imprecisioni nell'immagine iniziale: si pensi ad esempio un granello di polvere che può essere scambiato per un punto, o ad una "n" battuta male che viene scambiata per la sequenza "ii". Si possono distinguere le seguenti categorie di errore:

- **Word Error.** Aggiunta, rimozione o sostituzione di caratteri spuri all'interno di una parola. Questo tipo di errore si divide in due ulteriori sottocategorie:
  - Non-word error (NW): la parola affetta da errore non è presente in un dato vocabolario. Un errore di questo tipo è la parola "gioia" che diventa "g1o1a".
  - Real-word error (RW): la parola affetta da errore è presente in un dato vocabolario, ma non è corretta nella frase in cui è inserita. Ad esempio, la frase "qualvolta i popoli ripongono la loro fiducia nelle armi e nella guerra" può essere erroneamente interpretata come "qualvolta i popoli ripongono la loro fiducia nelle ami e nella guerra". "ami" è una parola presente nel vocabolario, ma non è corretta nel contesto della frase data.
- **Word Segmentation Error.** Si ha un errore di segmentazione quando aggiunte, rimozioni o sostituzioni di caratteri (incluso il carattere spazio) occorrono in maniera tale da dividere o unire parole. Sono considerati errori di segmentazione anche aggiunte di caratteri e segni di punteggiatura spuri all'interno del testo. Sono distinte le seguenti sottocategorie di errori di segmentazione:
  - Space Splitting (SP), quando i caratteri di una parola o più parole adiacenti sono intervallati da spaziature. Ad esempio, si ha un errore di questo tipo nella frase "vostra presenza conferma l'attaccamento alla cattedra di Pietro e la fedeltà al suo Magistero" che viene letta come

*"vostra presenza conferma l'a t t a c c a m e n t o a l l a cattedra di Pietro e la fedeltà al suo Magistero".*

- Punctuation Splitting (DP), quando uno o più segni di punteggiatura dividono in più parti una parola. Ad esempio, data la parola *"attaccamento"* si ha un errore DP quando essa viene interpretata come *"at,tacc,amento"*.
- Punctuation Insertion (PS), quando uno o più segni di punteggiatura vengono aggiunti fra una parola e l'altra. Ad esempio, data la frase *"Per secoli la Chiesa ha patrocinato artisti che hanno..."*, si hanno errori PS quando essa viene interpretata come *"Per secoli'la Chiesa ha.,patrocinato artisti che hanno..."*.

In Tabella 17 sono riportati ulteriori esempi degli errori appena elencati.

Originale	OCR	Tipo	Errore
dell'impresa	dell'iimpresa	NW	Sostituzione di "m" con "in"
interessano	iateressano	NW	Sostituzione di "n" con "a"
questo modo	questo nodo	RW	Sostituzione di "m" con "n"
l'ingordizia	l'ingordizia	RW	Sostituzione di "l" con "I"
produttrice	pro d u t t rice	SP	Parola spezzettata da spazi
azionisti	azi:misti	DP	Sostituzione con divisione della parola
vicinanze del	vicinanze .del	PS	Introduzione di punteggiatura

Tabella 17: Esempi di errori

Se, come già detto, gli errori derivano spesso da piccoli difetti nell'immagine di partenza, è lecito aspettarsi una maggior quantità di errori da documenti più deteriorati o datati, come archivi storici. Per lo stesso motivo l'intensità degli errori può variare, a seconda delle condizioni in parti diverse di uno documento.

**OCR Post-processing** Una strategia spesso adottata per minimizzare il numero di errori è l'aggiunta di una fase di post elaborazione (OCR post-processing) in seguito all'acquisizione dei documenti tramite OCR. Lo scopo della fase di OCR post-processing è quello di correggere gli errori introdotti durante l'estrazione del testo dalle immagini. Una delle difficoltà in questa fase sta nel non introdurre nuovi errori nel testo. Si pensi ai seguenti casi:

- Il sistema prova a correggere una parola che non contiene errori. In questo caso, qualsiasi correzione risulterà in un nuovo errore. Una delle maggiori criticità nello sviluppare il sistema di correzione, infatti, sta nell'identificare correttamente gli errori all'interno del testo.

- Il sistema identifica correttamente un errore, ma propone una correzione sbagliata. Si pensi al seguente esempio: "*vesodvi*" viene erroneamente corretto in "*vedovi*". Tuttavia, la parola originale era "*vescovi*".

La metodologia di correzione sviluppata è modellata come una pipeline che si compone di una serie di moduli ripetibili in sequenza. Ogni modulo è una funzione che si occupa di individuare e correggere una specifica categoria di errore.

La metodologia sviluppata si propone di correggere gli errori di tipo RW e SP.

## 3.2 Architettura del sistema

### 3.2.1 Panoramica Generale

Il sistema di correzione è articolato come una pipeline composta da uno o più moduli concatenati. Un modulo è una funzione il cui scopo è correggere una specifica tipologia di errore all'interno di una frase. Un modulo può essere formalmente definito come segue:

$$M : F \mapsto F' \quad (42)$$

dove  $F$  e  $F'$  sono rispettivamente la frase originale e quella con delle correzioni apportate.

Sono stati definiti i seguenti moduli:

- **Modulo di correzione Token** ( $M_{TK}$ ): individua e corregge tutti gli errori di tipo RW. È discusso in dettaglio nella sottosezione 3.2.3.
- **Modulo di correzione Split** ( $M_{SP}$ ): individua e corregge tutti gli errori di tipo SP. È discusso in dettaglio nella sottosezione 3.2.4.

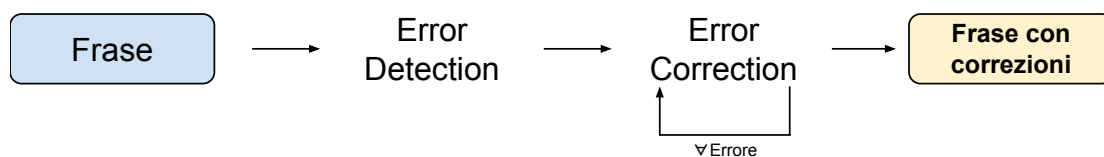


Figura 6: Schema del funzionamento di un modulo

Come si può vedere in Figura 6, il funzionamento di un modulo si compone delle fasi Error Detection (che da qui in poi sarà abbreviata con ED) ed Error Correction (che da qui in poi sarà abbreviata con EC). Durante la fase di ED vengono individuati gli errori presenti all'interno della frase da correggere: dato



che moduli differenti correggono tipologie di errore differenti, ogni modulo applica una diversa strategia per individuare gli errori. Nella fase di EC vengono effettuate le correzioni degli errori individuati nella fase precedente.

**Esempio** Si supponga di voler applicare alla seguente frase il modulo di correzione token:

*"Q o a l c h n Papa nort è compreso in questa comunità d1 morti gluriosi"*

Il modulo di correzione token riconosce e corregge solo gli errori di tipo NW (Non-word error, sottolineati con linea continua), mentre ignora tutti gli errori di altro tipo (sottolineati con linea tratteggiata). L'output del modulo, supponendo di riuscire a correggere tutti gli errori, è il seguente:

*"Q o a l c h n Papa non è compreso in questa comunità di morti gloriosi"*

Data la definizione di modulo, è possibile definire una pipeline di correzione come una concatenazione di uno o più moduli. Più formalmente, una pipeline di correzione è definita come segue:

$$P : f_1 \circ f_2 \circ \dots \circ f_n \quad (43)$$

dove ogni  $f_i$  è un modulo.



Figura 7: Schema riassuntivo della pipeline di correzione

**Esempio** Continuando l'esempio precedente, si supponga di concatenare un modulo  $M_{SP}$  al precedente modulo  $M_{TK}$ . Il modulo riceve come input la frase con le correzioni apportate dal modulo precedente, e corregge tutti gli errori di tipo SP presenti nella frase (ovvero quelli con la sottolineatura tratteggiata):

*"Qualche Papa non è compreso in questa comunità di morti gloriosi"*

All'interno della pipeline di correzione possono essere presenti più occorrenze di uno stesso modulo. In frasi con molti errori una sola applicazione di un dato modulo di correzione potrebbe non essere sufficiente per correggere tutti gli errori

della propria tipologia. Per ragioni legate al funzionamento del BERT Masked Language Model che verranno meglio chiarite nelle prossime sottosezioni, infatti, ripetere un dato modulo più volte può aumentare tangibilmente le performance di correzione.

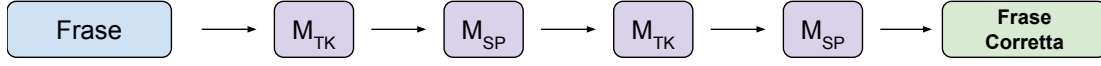


Figura 8: Esempio di una possibile pipeline con più occorrenze di uno stesso modulo

### 3.2.2 BERT Masked Language Modeling

La fase di error correction di entrambi i moduli implementati fa uso del BERT Masked Language Modeling (da qui in poi riferito come BERT MLM). Data una qualsiasi sequenza  $S = [w_1, \dots, w_n]$  nella quale ogni  $w_i$  è una parola, è possibile mascherare una sola parola  $w_i$  con la stringa  $[MASK]$ . Si ottiene così la sequenza con maschera  $S' = [w_1, \dots, [MASK], \dots, w_n]$ .

Dando in input  $S'$  al modello BERT, esso associa ad ogni parola del proprio lessico la probabilità di corrispondere alla parola mascherata. Le prime  $n$  parole con la probabilità più alta sono dette candidati. È quindi formalizzata come segue la funzione del BERT MLM detta "mask-filling":

$$B : S' \rightarrow [c_1, \dots, c_n] \quad (44)$$

dove ogni  $c_i \in [c_1, \dots, c_n]$  è un candidato.

**Esempio** Data la frase

*"che assistono ragazze in difficoltà, le persone soie e abbandonate, gli ammalati e gli anziani."*

la parola "soie" sottolineata è stata individuata come errore. È quindi necessario mascherarla, per dare la frase in input al modello BERT. La frase diventa dunque:

*"che assistono ragazze in difficoltà, le persone [MASK] e abbandonate, gli ammalati e gli anziani."*

BERT produce quindi una lista di candidati. Sono riportati i risultati impostando come soglia  $n = 5$ . I candidati rappresentano le top-5 più probabili correzioni.

- *"sole"* con probabilità 0.42
- *"anziane"* con probabilità 0.28
- *"povere"* con probabilità 0.08
- *"care"* con probabilità 0.03
- *"disabili"* con probabilità 0.01

Bisogna sottolineare come la parola originale sia trasparente al modello BERT. Ciò significa che i candidati prodotti dal modello sono del tutto indipendenti dalla parola originale, e sono inferiti unicamente dal contesto derivato dal resto della frase.

### 3.2.3 Modulo di correzione Token

#### Error Detection

Il modulo di correzione token ha lo scopo di individuare e correggere tutti gli errori di tipo NW. L'individuazione degli errori è effettuata in modo automatico dopo una pre-elaborazione (pre-processing) del testo. Il funzionamento di questa fase può quindi essere suddiviso in due stadi:

1. **Sentence pre-processing:** la frase viene tokenizzata, ovvero viene divisa in singole parole dette token. Questo stadio è implementato tramite la libreria NLTK[25].
2. **Error marking:** ogni token è analizzato singolarmente per determinare se contenga o meno un errore. Un token è considerato errato se sono vere le seguenti condizioni:
  - Ha una lunghezza minima di 2 caratteri. Questa condizione è necessaria per evitare di introdurre nuovi errori correggendo inutilmente errori di tipo SP. Ad esempio, sarebbe deleterio provare a correggere le singole lettere di *"Q u a l c h e"*. Ciò è dovuto al fatto che il modulo di correzione token è progettato per correggere singoli token: nell'esempio precedente, si tenterebbe quindi la correzione di tutte le sub-word che compongono il token. Per questo tipo di errori, è stato sviluppato il modulo di correzione Split (sottosezione 3.2.4).

- Dato un vocabolario di parole corrette, il token considerato non è presente in tale vocabolario. La ricerca nel vocabolario è eseguita in maniera case-insensitive. Tale approccio, seppur molto semplice, ha alcune criticità. Ad esempio, ogni parola non appartenente alla lingua italiana verrà segnalata come errore. È quindi auspicabile disporre di un vocabolario che comprenda anche alcune parole straniere di uso comune.

Un'eccezione a quest'ultima condizione è data da tutti i token che precedono un apostrofo. Questi token sono considerati corretti se almeno una fra tutte le combinazioni token + vocale è corretta. Si prenda ad esempio il token *"dell"* non presente nel vocabolario, seguito dal token apostrofo. Vengono generate le 5 combinazioni token-vocale *"della"*, *"delle"*, *"delli"*, *"dello"*, *"dellu"*. Siccome almeno una di queste è corretta, il token è considerato corretto.

Il processo appena descritto è schematizzato in Figura 9. Tutti gli errori identificati sono poi corretti in sequenza dalla fase di error correction.

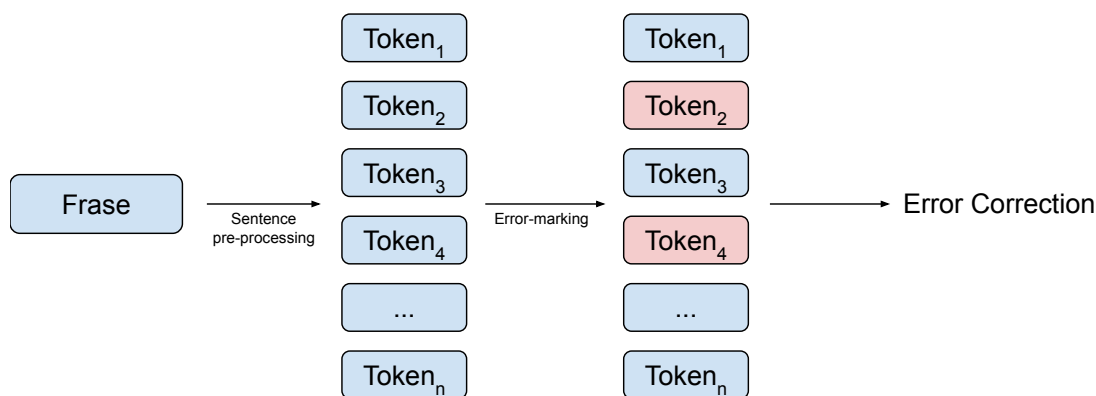


Figura 9: Schema del processo di error detection

### Error Correction

La fase di error correction ha lo scopo di correggere gli errori individuati nella precedente fase. La fase di error correction è composta dai seguenti stadi:

1. Masking
2. Detokenization
3. Candidates generation and picking
4. Validation

Ognuno di questi stadi è ripetuto per ognuno degli errori identificati durante la fase precedente.

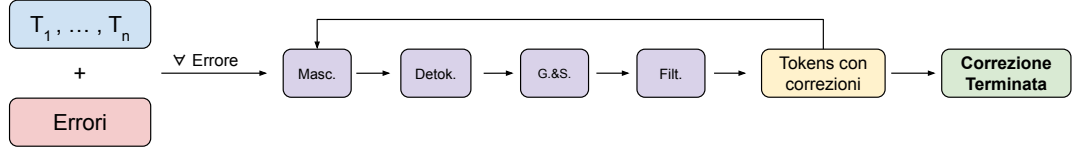


Figura 10: Schema della fase di error correction

**Masking** Lo stadio di masking ha lo scopo di mascherare uno dei token errati individuati nella fase precedente. Si ricorda che, a causa del funzionamento del BERT MLM descritto nella sottosezione 3.2.2, è possibile mascherare solo un token alla volta.

Data una sequenza di token  $S = [t_1, \dots, t_n]$  e l'insieme  $E = [t_i, t_j, \dots] \subseteq S$  dei token contenenti errore non ancora corretti, lo stadio di masking è descritto dalla seguente funzione:

$$M : S \rightarrow [t_1, \dots, [MASK], \dots, t_n] = S' \quad (45)$$

dove  $[MASK]$  sostituisce il primo token estratto dall'insieme  $E$ . Quanto descritto corrisponde al primo passaggio nella Figura 11.

**Detokenization** Come spiegato nella sottosezione 3.2.2, la funzione di mask-filling del BERT MLM richiede come input una sequenza intesa come frase, e non come insieme di token. È quindi compito dello stadio di detokenization ricomporre una sequenza a partire dall'output dello stadio precedente. Lo stadio di detokenization è descritto dalla seguente funzione:

$$D : [t_1, \dots, [MASK], \dots, t_n] \rightarrow F_{mask} \quad (46)$$

dove  $F_{mask}$  è una frase contenente una maschera.

**Candidates generation and picking** Lo stadio di candidates generation and picking sfrutta il BERT MLM per generare dei candidati per la correzione, e sfrutta un'euristica per determinare il candidato con la maggior probabilità di essere corretto (da qui in poi riferito come soluzione). Più precisamente, la generazione dei candidati avviene tramite la funzione di mask-filling descritta nell'Equazione 44 (sottosezione 3.2.2). La fattibilità di applicare tale approccio per la generazione di

candidati per la correzione è dimostrata in [23]. Nel paper appena citato si riporta come, usando una combinazione di BERT e FastText, la giusta correzione da applicare sia presente fra i candidati prodotti con una probabilità del 70%. Risultati comparabili sono stati ottenuti con una metodologia simile dai test illustrati nella Capitolo 5.

Lo step di generazione dei candidati è formalizzato come segue:

$$G : F_{mask} \rightarrow [c_1, \dots, c_n] = C \quad (47)$$

dove ogni  $c_i$  è un candidato.

L'euristica per la scelta della soluzione sceglie il candidato con la più bassa distanza di Levenshtein dal token mascherato. Più formalmente, chiamando  $m$  il token mascherato, la soluzione  $s$  è definita come segue:

$$s = \underset{c_i \in C}{\operatorname{argmin}} d_{lev}(c_i, m) \quad (48)$$

In caso uno o più candidati abbiano la stessa distanza dal token mascherato, viene scelto quello con la maggior probabilità prodotta da BERT. Questa euristica, seppur molto semplice, è decisamente efficace, come dimostrato dall'analisi nella Sezione 5.4.

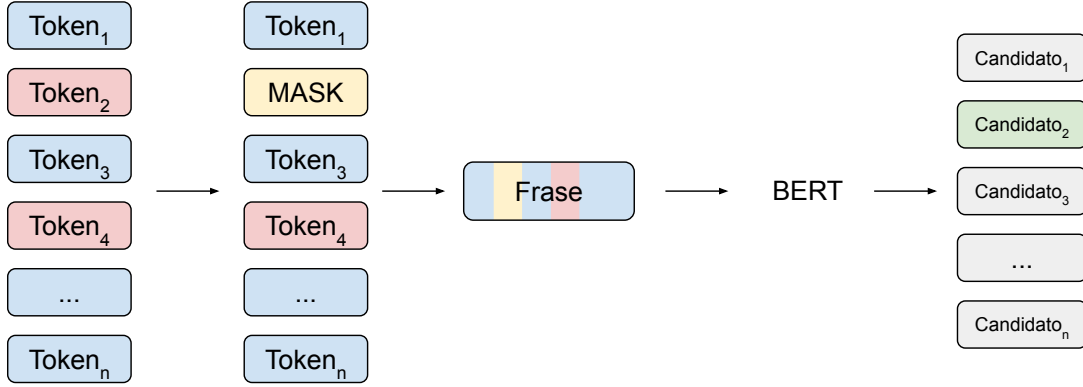


Figura 11: Schema del processo di generazione e scelta dei candidati

**Validation** Può accadere che la soluzione non sia una correzione adatta: si pensi ai seguenti casi:

1. L'error detection contrassegna un token corretto come errore. In questo caso, ogni tentativo di correzione introdurrebbe nuovi errori all'interno del testo.

2. BERT non produce la giusta correzione fra i candidati. Anche in questo caso, qualunque sia il candidato scelto, il sistema di correzione andrebbe a introdurre nuovi errori all'interno del testo.

È introdotta quindi un'ulteriore fase di validation, con lo scopo di valutare se la soluzione prodotta possa o meno rappresentare una correzione valida. La validazione è implementata tramite un'euristica che valuta se la soluzione trovata è troppo lontana (in termini di distanza di Levenshtein) dal token mascherato. Chiamando  $l$  la lunghezza in caratteri del token mascherato, e  $d$  la distanza di Levenshtein fra il token mascherato e la soluzione trovata, la soluzione è valida se:

- $l > 10 \wedge d < 5$
- $l > 5 \wedge d < 4$
- $l \leq 5 \wedge d < 3$

Se la validazione scarta la soluzione trovata il sistema ignora la correzione. Al contrario, se la validazione ritorna esito positivo, la correzione viene sostituita al token mascherato. Quanto appena descritto è rappresentato in Figura 12.

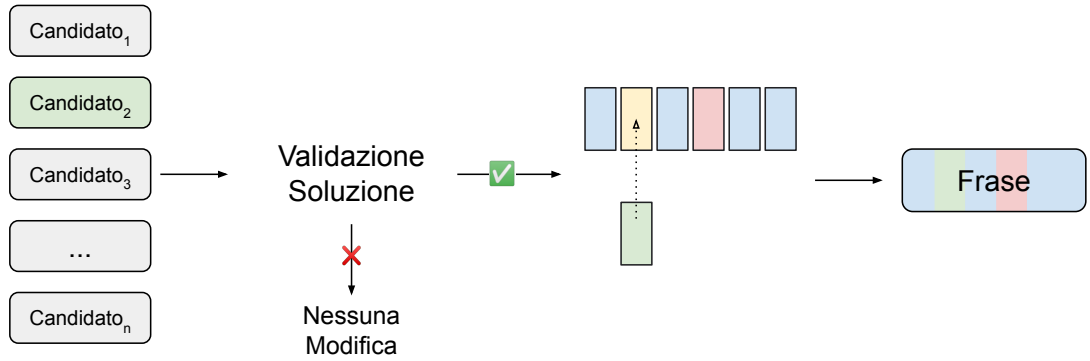


Figura 12: Schema del processo di scelta dei candidati

Il processo appena descritto si ripete per ogni errore contrassegnato durante la fase di error detection. Una volta completata la correzione dell'ultimo token errato, la fase di error correction può dirsi conclusa.

### Ripetizione

In frasi contenenti molti errori, è possibile che una sola applicazione del modulo di correzione token non basti correggere tutti i non-word errors.

Ciò è dovuto al funzionamento di BERT, che usa il contesto a destra e sinistra della

maschera per produrre i candidati di correzione. Se questo contesto è sporcato da molti errori, è più probabile che BERT non sia in grado di produrre una soluzione adeguata. È quindi possibile che dopo l'applicazione di questo o altri moduli, e la conseguente correzione di alcuni errori, sia possibile ottenere migliori risultati sulle correzioni precedentemente non riuscite.

Questo approccio comporta però uno svantaggio: se la prima o una delle prime correzioni sono sbagliate, il rischio è quello di introdurre rumore anche nelle correzioni successive.

### 3.2.4 Modulo di correzione Split

#### Error detection

Il modulo di correzione Split corregge tutti gli errori di tipo SP.

La fase di error detection si compone di tre stadi:

1. **Sequences marking:** data una frase contenente errori di tipo SP, sono delimitati i punti di inizio e terminazione delle sequenze contenenti errori. L'individuazione di tali sequenze è eseguita tramite la seguente espressione regolare:

$$r' (?: \backslash s | ^) (\backslash w (?: \backslash W? \backslash s \backslash w \{ 1, 2 \} ) \{ 3, \} ) (?: \backslash W | \backslash s | \$) '$$

La precedente espressione regolare intercetta tutte le sequenze di almeno tre gruppi di caratteri alfanumerici con lunghezza massima due intervallati da spazi e/o caratteri non alfanumerici. Per maggiore chiarezza, in Tabella 18 sono riportati alcuni esempi di sequenze riconosciute.

2. **Punctuation filtering:** all'interno delle sequenze vengono rimossi tutti i caratteri non alfanumerici o spaziature. Ciò serve a rimuovere eventuali segni di punteggiatura spuri che possono diminuire l'efficacia delle seguenti fasi.
3. **Error validation:** non tutte le sequenze riconosciute dall'espressione regolare sono degli errori. Si pensi ad esempio ad una serie di preposizioni o congiunzioni, come il quarto esempio in Tabella 18. Si usa quindi un'euristica per scartare le sequenze che si ritiene non contengano errori. Sono considerati errori tutte le sequenze nelle quali il numero di gruppi di 2 caratteri è minore del numero di gruppi di 1 carattere.



#	Frase	Sequenza	Errore
1	" <i>d i l e t t i membri, della ...</i> "	<i>d i l e t t i</i>	Sì
2	" <i>guidato d u i l l a fede</i> "	<i>d u i l l a</i>	Sì
3	" <i>... altro tipo di u n i o, n e.</i> "	<i>u n i o, n e</i>	Sì
4	" <i>in me e io in te, siano anch...</i> "	<i>in me e io in te</i>	No
5	" <i>...dopo qu al ch e esitazione...</i> "	<i>qu al ch e</i>	Sì
6	" <i>...dopo q u al che esitazione...</i> "	<i>q u al</i>	Sì
7	" <i>...dopo qu al che esitazione...</i> "	/	/

Tabella 18: Esempi di sequenze riconosciute

L'approccio di error correction appena esposto presenta però alcune limitazioni. Se uno dei gruppi della parola divisa contiene più di tre caratteri alfanumerici, la sequenza prodotta può:

- Non essere riconosciuta. È il caso dell'esempio 7 nella Tabella 18. In questo caso, il gruppo "*che*" non è catturato dall'espressione regolare. Ciò comporta che il numero di gruppi di caratteri sia troppo basso per essere riconosciuto dall'espressione regolare.
- Essere riconosciuta parzialmente. È il caso dell'esempio 6 nella Tabella 18. In questo caso, il gruppo "*che*" non è catturato dall'espressione regolare. Ciò comporta che venga catturata la sequenza "*q u a l*", che verrà corretta, introducendo possibilmente un errore. Il gruppo "*che*", invece, viene erroneamente considerato corretto.

Data quindi una frase  $f \in F$  contenente errori di segmentazione, dove  $F$  è l'insieme di tutte le frasi, la fase di error detection è formalizzata come segue:

$$ED_{split} : F \rightarrow L_{seq} \quad (49)$$

dove ogni  $l_{seq} \in L_{seq}$  è un insieme di tuple  $[(s_1, b_1, e_1), \dots, (s_n, b_n, e_n)]$  nel quale ogni tupla corrisponde ad un errore individuato e validato. Ogni tupla è composta come segue:

- $s_i$  è la sequenza filtrata contenente l'errore
- $b_i$  è il punto di inizio in  $f$  della sequenza
- $e_i$  è il punto di fine in  $f$  delle sequenze

**Esempio** Data la frase:

*coinvolsero ogni uomo e o' g n i donna, p r o c u r a n d o.*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
c	o	i	n	v	o	l	s	e	r	o		o	g	n	i		u	o	m

20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
o		e		o	'		g		n		i		d	o	n	n	a	,	

40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
p		r		o		c		u		r		a		n		d		o	.

Tabella 19: Caratteri e indici della frase di esempio

si procede con lo stadio di sequences marking. Le sequenze individuate sono quelle sottolineate nella frase data. Sono riportate di seguito insieme alla loro posizione di inizio e di fine.

#	Sequenza	Pos.Inizio	Pos.Fine
1	"e o' g n i"	22	32
2	"p r o c u r a n d o"	40	59

Si procede quindi con lo stadio di punctuation filtering:

#	Sequenza	Pos.Inizio	Pos.Fine
1	"e o g n i"	22	32
2	"p r o c u r a n d o"	40	59

Si noti come nella prima sequenza è stato cancellato un apostrofo, ma le posizioni di inizio e fine rimangono le stesse, in quanto demarcano l'inizio e la fine della sequenza nella frase originale. Lo stadio di error validation in questo caso non scarta alcuna soluzione: in entrambe le sequenze, infatti, il numero di gruppi di 1 carattere è maggiore del numero di gruppi di 2 caratteri.

### Error correction

La fase di Error Correction mira a correggere gli errori individuati nella fase precedente. Più formalmente è definita come:

$$EC_{split} : (F, L_{seq}) \rightarrow F_{corr} \quad (50)$$

dove  $F_{corr}$  è la frase con le correzioni apportate.

La fase di Error Correction si compone dei seguenti stadi:

1. **Compression:** per ogni sequenza  $s_i$  individuata da correggere, si rimuovono gli spazi bianchi, in modo da ottenere un'unica parola continua detta  $s'_i$ .
2. **Vocabulary correction:** se una sequenza  $s'_i$  è presente nel vocabolario, allora  $s'_i$  viene inserita nella frase al posto della sequenza contenente l'errore.
3. **BERT correction:** si fa uso del BERT MLM per trovare una correzione, in modo simile a quanto fatto nel modulo di correzione token. A seconda della tipologia dell'errore sono applicabili due strategie di correzione.
  - Con sottrazione
  - Con rimpiazzo

Ognuno degli stadi appena descritti è applicato in sequenza ad ogni singola tupla prodotta dalla fase precedente.

**Funzione di sostituzione** Viene introdotta una funzione detta funzione di sostituzione. Data una frase  $f \in F$ , una sequenza  $s \in S$ , un valore di inizio  $b \in I$  e un valore di fine  $e \in E$ , la funzione è definita come:

$$Sos : (F, S, I, E) \rightarrow F' \quad (51)$$

dove  $F'$  è la frase  $F$  in cui tutti i caratteri fra  $b$  ed  $e$  sono stati sostituiti da  $s$ . Alternativamente, definendo come  $T$  la tupla  $(s, b, e)$  è possibile scrivere la funzione anche come segue:

$$Sos : (F, T) \rightarrow F' \quad (52)$$

Ad esempio, data la frase  $f_{es}$ :

*Onorando una tradizione plurisecolare*

e la tupla  $t_{es}$  ("tradizione", 14, 25), si ha che:

$$Sos(f_{es}, t_{es}) = \text{Onorando una tradizione plurisecolare} \quad (53)$$

**Compression** Lo stadio di compression rimuove gli spazi da ogni sequenza  $s$  individuata come errore. Più formalmente, data una tupla  $(s, b, e) \in T$ , lo stadio di compression è definito come segue:

$$Comp : T \rightarrow T' \quad (54)$$

dove  $T'$  è l'insieme di tutte le tuple  $(s', b, e)$  in cui  $s'$  è la sequenza  $s$  senza spaziature. Riprendendo l'esempio proposto nella fase di error detection, le tuple prodotte diventano rispettivamente:

#	Sequenza	Pos.Inizio	Pos.Fine
1	"eogni"	22	32
2	"procurando"	40	59

**Vocabulary correction** Date le tuple prodotte dallo stadio precedente, lo stadio di vocabulary correction applica una correzione per tutte le tuple la cui sequenza corrisponde ad una parola presente nel vocabolario.

Data quindi la frase  $F$  contenente errori di segmentazione e una tupla  $(s', b, e)$  detta  $t_{corr}$  che rispetta la condizione enunciata precedentemente, la frase con la correzione applicata  $F_{corr}$  si ottiene come segue:

$$F_{corr} = Sos(F, t_{corr}) \quad (55)$$

Se la tupla che si sta trattando può essere utilizzata per una correzione con vocabolario, la correzione si arresta in questo stadio. Altrimenti, si procede allo stadio di BERT correction.

Continuando l'esempio predente, si nota come la seconda tupla contenga la sequenza "procurando" che è una parola all'interno del vocabolario. Ricordando la frase originale

*coinvolsero ogni uomo e o' g n i donna, p r o c u r a n d o.*

detta  $F_{ex}$ , e la tupla  $t_2$  ("procurando", 40, 59), la correzione si applica come segue:

$$F_{corr} = Sos(F_{ex}, t_2) \quad (56)$$

Il risulta è la frase  $F_{corr}$ :

*coinvolsero ogni uomo e o' g n i donna, procurando.*

**BERT correction** Per tutte le tuple per le quali non è stato possibile applicare una correzione con vocabolario, si passa allo stadio di BERT correction. Il primo

step di questo stadio consiste nel mascheramento della sequenza errata nella frase originale. Data una tupla  $(s', b, e)$  prodotta dallo stadio di compressione e una frase  $F$ , si ottiene la frase mascherata  $F_{mask}$  come segue:

$$F_{mask} = Sos(F, [MASK], b, e) \quad (57)$$

A questo punto si sfrutta il BERT MLM per produrre un candidato per la correzione. Il processo è analogo alla fase di generazione e scelta dei candidati del modulo di correzione token nella Sezione 3.2.3, per cui si rimanda il lettore al paragrafo che ne spiega il funzionamento.

Una volta prodotto il candidato per la correzione, detto  $c_{corr}$ , è possibile applicare una correzione per sottrazione o per rimpiazzo.

**Correzione con sottrazione** Si applica in caso  $c_{corr}$  coincida con la fine o l'inizio della sequenza compressa  $s'$ . Questo è il caso in cui la sequenza riconosciuta come errore di spezzettamento con spazi comprende due parole. Pertanto, se  $c_{corr}$  corrisponde all'inizio di  $s'$ , si ottengono le due stringhe  $sub_1$  e  $sub_2$  come segue:

$$sub_1, sub_2 = c_{corr}, s' - c_{corr} \quad (58)$$

Se invece  $c_{corr}$  corrisponde alla fine di  $s'$ , le stringhe  $sub_1$  e  $sub_2$  si ottengono come segue:

$$sub_1, sub_2 = s' - c_{corr}, c_{corr}, \quad (59)$$

Data quindi la frase  $F$  contenente errori di segmentazione, la tupla  $(s', b, e)$  e le stringhe  $sub_1, sub_2$ , la frase con la correzione applicata  $F_{corr}$  si ottiene come segue:

$$F_{corr} = Sos(F, sub_1 + " " + sub_2, b, e) \quad (60)$$

La correzione con sottrazione corregge i casi in cui la sequenza contenente errore è formata da due parole. Può accadere che la stringa  $s' - c_{corr}$  non rappresenti una parola corretta, ovvero potrebbe non essere presente nel vocabolario. Non sono previsti controlli e contromisure ad hoc per tale eventualità: la correzione di eventuali errori di questo tipo è delegata ai moduli di correzione token che possono essere inseriti a valle nella pipeline.

Si consideri ora l'esempio portato avanti negli stadi precedenti: data la tupla  $t_2$  ("*eogni*", 22, 32) e la frase  $F_{ex}$

*coinvolsero ogni uomo e o' g n i donna, p r o c u r a n d o.*

si ottiene come la frase  $F_{mask}$  applicando la funzione di sostituzione:

$$F_{mask} = Sos(F_{ex}, [MASK], 22, 32) \quad (61)$$

Si ottiene dunque la frase

*coinvolsero ogni uomo [MASK] donna, p r o c u r a n d o.*

Sfruttando il BERT MLM si ottiene il candidato  $c_{corr}$  "ogni". Il candidato corrisponde alla fine di  $s'$ , pertanto si ottengono  $sub_1, sub_2 = e, ogni$ . Si applica quindi la funzione di sostituzione:

$$F_{corr} = Sos(F_{ex}, e + " " + ogni, 22, 32) \quad (62)$$

La frase corretta  $F_{corr}$  è quindi:

*coinvolsero ogni uomo e ogni donna, p r o c u r a n d o.*

**Correzione con rimpiazzo** Si applica in caso  $c_{corr}$  non coincida con la fine o l'inizio della sequenza compressa  $s'$ . Questo step utilizza un'euristica per validare la correzione prodotta dal BERT MLM. La soluzione prodotta da BERT è valutata ammissibile se e solo se  $d_{lev}(s', c_{corr}) \leq 4$ .

In caso la soluzione sia ammissibile, la correzione è applicata tramite la funzione di sostituzione:

$$F_{corr} = Sos(F, c_{corr}, b, r) \quad (63)$$

In caso la soluzione non sia ammissibile, la correzione si arresta senza apportare correzioni.

# Capitolo 4

## Test e Risultati

In questa sezione sono illustrate la metodologia di test, le metriche utilizzate e risultati ottenuti dai test svolti.

Nella sezione Sezione 4.1 sono illustrati gli obiettivi dei test e la metodologia con la quale sono svolti. Nella Sezione 4.2 sono descritte le metriche usate per svolgere i test. Nella Sezione 4.3 sono riportati i risultati ottenuti dai test.

### 4.1 Introduzione

La metodologia di test adottata ha lo scopo di valutare automaticamente e in modo ripetibile le prestazioni del sistema di correzione. Tale metodologia è utile durante lo sviluppo del sistema di correzione, in quanto rende possibile quantificare l'impatto delle modifiche apportate al funzionamento e alla configurazione.

Inoltre, i test svolti in questo capitolo rendono possibile:

- Valutare le performance del sistema su vari tipi e intensità di errore. Come illustrato nel Capitolo 2, per ogni frase perturbata nel dataset, sono presenti 9 versioni perturbate con diverse superpipeline. Valutando le performance di correzione sulle diverse tipologie di frasi, è possibile determinare come si comporta il sistema di correzioni di fronte a diversi tipi di errore.
- Confrontare le performance del sistema di correzione su diverse versioni del dataset. Nel Capitolo 2 sono infatti state definite due versioni del dataset, `dst@50` e `dst@100`, che differiscono per la lunghezza delle frasi. Portare a termine i test su entrambe le versioni permette quindi di capire quanto la lunghezza delle frasi influisca sulle prestazioni del sistema di correzione.
- Confrontare il sistema di correzione sviluppato con altri sistemi di OCR post-processing per avere un riscontro con il quale meglio valutare i risultati

ottenuti. A tale scopo, si utilizza come confronto la metodologia di correzione proposta in [15], già illustrata nella sottosezione 1.1.2; l'approccio utilizzato è però leggermente modificato, e utilizza una combinazione di FastText e Word2Vec per trovare le parole semanticamente simili a quelle errate (a differenza del solo Word2Vec come nel paper originale). In questa sezione ci si riferirà con:

- **Bert** al metodo sviluppato e oggetto di questa tesi.
- **Ftwv** al metodo appena descritto usato come confronto.

**Metodologia** Sono dati:

- L'insieme  $D = \{dst@50, dst@100\}$  dei dataset
- L'insieme  $M = \{Bert, Ftwv\}$  dei metodi di correzione

Ogni frase  $f$  in  $dst@50$  o  $dst@100$  contiene il testo originale ( $text$ ), e le sue versioni perturbate nel campo  $perturbed$ . Dato il metodo  $m \in M$  che si vuole valutare, e data una delle versioni perturbate  $v$  della frase nel campo  $perturbed$ , si produce la tripla in Tabella 20.

Nome Campo	Descrizione
$t_{or}$	Testo originale ( $text$ )
$t_{pe}$	Testo perturbato ( $v$ )
$t_{co}$	$v$ corretta con il metodo $m$

Tabella 20: Tripla contenente la frase corretta con uno dei metodi in  $M$

Dato un dataset  $d \in D$ , si producono 9 di queste triple per ogni frase  $f \in d$ , una per ognuna delle versioni perturbate presenti nel campo  $perturbed$  di  $f$ . Le triple sono raggruppate in 9 insiemi (T1, T2, T3, S1, S2, S3, M1, M2, M3) in base alla superpipeline di riferimento del campo  $t_{pe}$ . Su ognuno di questi insiemi è quindi possibile calcolare le metriche illustrate nella Sezione 4.2.

## 4.2 Metriche

La valutazione della correzione avviene attraverso le metriche definite in questa sezione. Le metriche definite sono:

1. Errori corretti per errore presente ( $C/P$ ).
2. Errori introdotti per carattere ( $In/Ch$ ).



3. Errori introdotti per errore corretto ( $I/C$ ).
4. Riduzione nella distanza di Levenshtein. ( $LDR$ ).
5. Distanza di Levenshtein totale ( $LDT$ ).

Prima di spiegare nel dettaglio le metriche introdotte, è spiegato il concetto di "errore delimitato" usato nelle prime tre metriche.

**Errore delimitato** Siano date:

- $s$ : una sequenza di testo considerata corretta.
- $s'$ : una sequenza di testo che differisce da  $s$  per alcuni errori.

Si definiscono "matching block" tutte le sottosequenze coincidenti di  $s$  e  $s'$ , attraverso le quali è anche possibile allineare le due sequenze. Si prendano ad esempio le sequenze "Adesso io preferisco" e "Addcso io prioferisco"; è possibile allinearne i matching block come segue:

<u>Ad</u>	<u>es</u>	<u>so</u>	<u>io</u>	<u>pr</u>	<u>e</u>	<u>ferisco</u>
<u>Ad</u>	<u>dc</u>	<u>so</u>	<u>io</u>	<u>pr</u>	<u>io</u>	<u>ferisco</u>

Si noti come i matching block siano stati sottolineati. Le sottosequenze che non sono comprese nei matching block, nella seconda frase dell'esempio "dc" e "io", sono quelle che contengono le differenze fra le due frasi, che in questo caso corrispondono agli errori. Queste sottosequenze sono chiamate *errori delimitati*.

Si prenda ad esempio la frase:

*"Adesso io preferisco parlare spontaneamente."*

In Tabella 21 sono date alcune versioni perturbate della stessa frase. Gli errori delimitati sono segnati fra parentesi quadre:

Esempio	Errori delimitati
A[, ]des[, ]so io preferisco parlare spontaneamente.	2
Ad[dc]so io pr[io]ferisco parlare spontaneamente.	2
Ad[c]sso[.] io pr[c]f[c]risc parlare spontaneamente.	3

Tabella 21: Versioni perturbate della frase di esempio con gli errori delimitati segnati dalle parentesi quadre

Si noti come ogni errore delimitato contribuisca al conteggio del numero di errori con un +1 indipendentemente dal numero di caratteri da cui è composto.

Data quindi una qualsiasi tripla  $t$  appartenente a uno dei 9 insiemi definiti in precedenza, e formata dai campi  $t_{or}$ ,  $t_{pe}$ ,  $t_{co}$ , sono date le seguenti definizioni:

- **Errori corretti:** numero di errori delimitati che sono in  $t_{pe}$  ma non in  $t_{co}$ . Più informalmente, è il numero di errori che il sistema di correzione utilizzato è riuscito a correggere.
- **Errori presenti:** numero di errori delimitati che sono in  $t_{pe}$ . Più informalmente, è il numero di errori che la superpipeline di perturbazione utilizzata ha generato in  $t_{pe}$ .
- **Errori introdotti:** numero di errori delimitati in  $t_{co}$  che non sono né in  $t_{pe}$ , né in  $t_{or}$ . Più informalmente, è il numero di errori che il sistema di correzione ha introdotto provando a correggere  $t_{pe}$ .

Data quindi una qualsiasi tripla  $t$  appartenente a uno dei 9 insiemi definiti in precedenza, sono definite le seguenti funzioni:

- $f_{co}(t)$ : data una tripla  $t$ , ne restituisce il numero di errori corretti.
- $f_{pr}(t)$ : data una tripla  $t$ , ne restituisce il numero di errori presenti.
- $f_{in}(t)$ : data una tripla  $t$ , ne restituisce il numero di errori introdotti.

Date queste definizioni, sono di seguito definite le metriche elencate in precedenza.

**Errori corretti per errore presente ( $C/P$ )** Lo scopo di questa metrica è quello di calcolare la percentuale di errori che il sistema di correzione riesce a risolvere. Dato un insieme  $I$  di triple  $(t_{or}, t_{pe}, t_{co})$ , è possibile definire  $C/P$  come segue:

$$C/P = \frac{\sum_{t \in I} f_{co}(t)}{\sum_{t \in I} f_{pr}(t)} \quad (64)$$

$C/P$  vale 0 se il sistema di correzione non corregge alcun errore, mentre vale 1 se il sistema di correzione li corregge tutti. È quindi chiaro come più alto è il valore di  $C/P$ , migliori sono le performance di correzione.

**Errori introdotti per frase ( $In/Ch$ )** Lo scopo di questa metrica è quello di quantificare gli errori che il sistema di correzione introduce. Dato un insieme  $I$  di triple  $(t_{or}, t_{pe}, t_{co})$ , è possibile definire  $In/Ch$  come segue:

$$In/Ch = \frac{\sum_{t \in I} f_{in}(t)}{|I|} \quad (65)$$

La metrica è divisa per il numero caratteri nel dataset, in modo tale da permettere anche la comparazione fra dataset con lunghezze differenti. In questo caso, valori più bassi di  $In/Ch$  sono desiderabili, in quanto si vuole evitare che il sistema di correzione non introduca nuovi errori.

**Errori introdotti per ogni errore corretto ( $I/C$ )** Lo scopo di questa metrica è quello di misurare il rapporto fra gli errori che il sistema di correzione introduce durante la correzione, e gli errori presenti che invece vengono corretti. Dato un insieme  $I$  di triple  $(t_{or}, t_{pe}, t_{co})$ , è possibile definire  $I/C$  come segue:

$$I/C = \frac{\sum_{t \in I} f_{in}(t)}{\sum_{t \in I} f_{co}(t)} \quad (66)$$

In questo caso, a valori più bassi di  $I/C$  corrispondono performance migliori del sistema di correzione. Più precisamente:

- Se  $I/C \geq 1$ , significa che il sistema di correzione sta introducendo più errori di quanti ne corregge, peggiorando nel complesso il testo in input.
- Se  $0 < I/C < 1$ , significa che il sistema di correzione sta correggendo più errori di quanti ne introduce, migliorando quindi il testo in input. Più il valore di  $I/C$  approssima lo 0, migliori sono le performance di correzione.

**Riduzione nella distanza di Levenshtein ( $LDR$ )** Lo scopo di questa metrica è quello di misurare quanto il processo di correzione avvicina la frase perturbata a quella originale. Si misura quindi la differenza in distanza di Levenshtein fra frase originale e frase perturbata e fra frase originale e frase corretta. Lo scarto fra queste distanze indica la riduzione in distanza. Dato un insieme  $I$  di triple  $(t_{or}, t_{pe}, t_{co})$ , è possibile definire  $LDR$  come segue:

$$LDR = \frac{\sum_{t \in I} d(t_{or}, t_{pe}) - d(t_{or}, t_{co})}{\sum_{t \in I} |t_{or}|} \quad (67)$$

Il valore ottenuto è diviso per la somma dei caratteri totali contenuti nell'insieme  $I$ . In questo modo la differenza è espressa nel rapporto riduzione su numero di caratteri, consentendo così di confrontare misurazioni su dataset differenti. Per questa metrica valori maggiori indicano migliori performance del sistema di correzione.

**Distanza di Levenshtein totale ( $LDT$ )** Lo scopo di questa metrica è quello di misurare la distanza totale fra le frasi originali e quelle corrette. Dato un insieme  $I$  di triple  $(t_{or}, t_{pe}, t_{co})$ , è possibile definire  $LDT$  come segue:

$$LDT = \frac{\sum_{t \in I} d(t_{or}, t_{co})}{\sum_{t \in I} |t_{or}|} \quad (68)$$

Il valore ottenuto è diviso per la somma dei caratteri totali contenuti nell'insieme  $I$ . In questo modo la differenza è espressa nel rapporto riduzione su numero di caratteri, consentendo così di confrontare misurazioni su dataset differenti. Per questa metrica valori più bassi indicano migliori performance del sistema di correzione.

### 4.3 Risultati

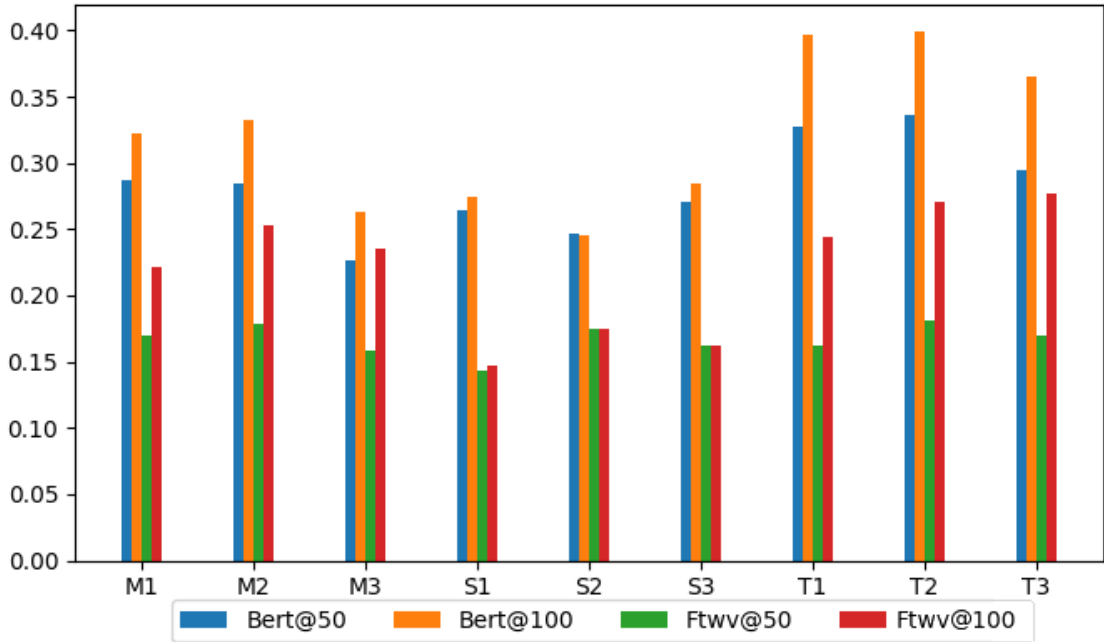
Lo scopo dei test in questa sezione è quello di valutare le performance del sistema di correzione sviluppato e confrontarle a quelle del sistema Ftwv. Si vuole inoltre misurare l'impatto della lunghezza delle frasi da correggere sulle performance di correzione. Per fare ciò, si eseguiranno i test sulle seguenti configurazioni:

- **Bert@50**: sistema di correzione Bert con dataset dst@50.
- **Bert@100**: sistema di correzione Bert con dataset dst@100.
- **Ftwv@50**: sistema di correzione Ftwv con dataset dst@50.
- **Ftwv@100**: sistema di correzione Ftwv con dataset dst@100.

**Errori corretti per errore presente ( $C/P$ )** In Tabella 22 e Figura 13 sono riportate le misurazioni per la metrica  $C/P$ . Dati i risultati, è possibile fare le seguenti osservazioni:

- Per ogni superpipeline, le configurazioni che usano frasi più lunghe ottengono risultati migliori rispetto alle controparti che utilizzano frasi da 50 caratteri. Questo fatto supporta l'ipotesi che la presenza di un contesto più ampio attorno a un errore possa aiutare durante la correzione.
- Il sistema di correzione sviluppato ottiene performance significativamente maggiori nelle token superpipeline. Questo era il risultato atteso, in quanto la correzione dei word error è stato il focus principale durante lo sviluppo del sistema.

Superpipeline	Bert@50	Ftwv@50	Bert@100	Ftwv@100
M1	0.29	0.17	0.32	0.22
M2	0.28	0.18	0.33	0.25
M3	0.23	0.16	0.26	0.24
S1	0.26	0.14	0.27	0.15
S2	0.25	0.17	0.25	0.17
S3	0.27	0.16	0.28	0.16
T1	0.33	0.16	0.4	0.24
T2	0.34	0.18	0.4	0.27
T3	0.29	0.17	0.37	0.28

Tabella 22: Misurazioni di  $C/P$  su tutti gli insiemi di frasiFigura 13: Misurazioni di  $C/P$  su tutti gli insiemi di frasi

**Errori introdotti per carattere ( $In/Ch$ )** In Tabella 23 e Figura 14 sono riportate le misurazioni per la metrica  $In/Ch$ . Dati i risultati si possono fare le seguenti osservazioni:

- In questo caso la lunghezza delle frasi non influenza i risultati ottenuti, che sono comparabili a parità di sistema di perturbazione.

- Si nota un ampio scarto fra i due sistemi di correzione: ciò in parte è attribuibile all'efficacia dello stadio di validazione durante la correzione nel sistema Bert. Non è però possibile escludere che il sistema Ftwv sia particolarmente pronò all'introduzione di errori durante la correzione.

Superpipeline	Bert@50	Ftwv@50	Bert@100	Ftwv@100
M1	0.0051	0.0226	0.0055	0.0245
M2	0.0056	0.0217	0.0055	0.0235
M3	0.0063	0.0206	0.0076	0.024
S1	0.0044	0.024	0.0042	0.0236
S2	0.0043	0.0235	0.0042	0.023
S3	0.0043	0.0219	0.0046	0.0212
T1	0.0049	0.0238	0.0048	0.0255
T2	0.0051	0.0232	0.005	0.0253
T3	0.0058	0.023	0.0056	0.0259

Tabella 23: Misurazioni di  $In/Ch$  su tutti gli insiemi di frasi

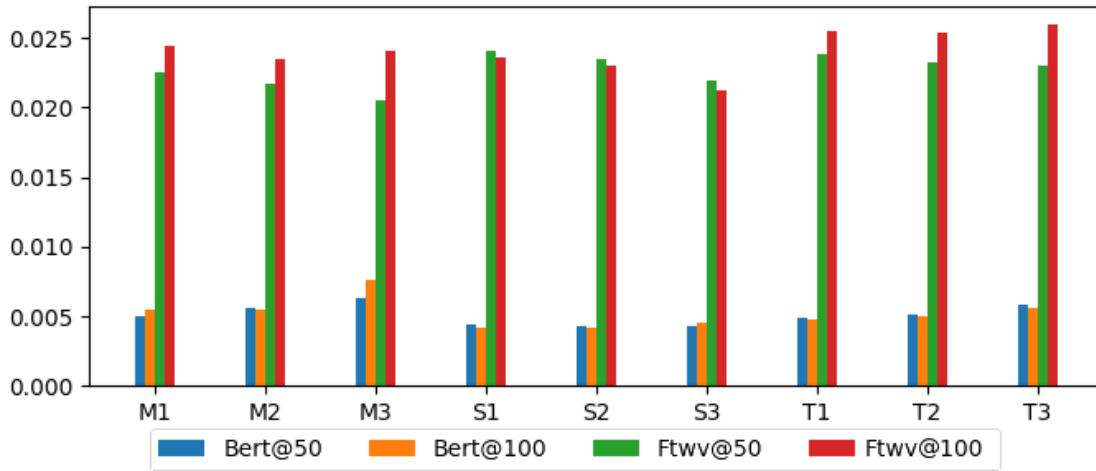


Figura 14: Misurazioni di  $In/Ch$  su tutti gli insiemi di frasi

**Errori introdotti per errore corretto ( $I/C$ )** In Tabella 24 e Figura 15 sono riportate le misurazioni per la metrica  $I/C$ . Dati i risultati si possono fare le seguenti osservazioni:

- Sono confermate le osservazioni fatte in precedenza: il sistema Bert corregge più errori e ne introduce di meno rispetto al sistema Ftwv. Con la sola

eccezione di S1, il sistema bert ha un  $I/C$  ampiamente inferiore a 1 su tutte le superpipeline e su entrambe le lunghezze di frasi. Ciò implica che il sistema corregge più errori di quanti ne introduce.

- Per questa metrica è presente una lieve ma non trascurabile differenza di prestazioni derivante dalla lunghezza delle frasi: frasi più lunghe, anche in questo caso, portano a risultati migliori.

Superpipeline	Bert@50	Ftwv@50	Bert@100	Ftwv@100
M1	0.36	2.79	0.35	2.3
M2	0.32	2.01	0.26	1.49
M3	0.29	1.37	0.3	1.05
S1	1.02	10.26	0.91	9.38
S2	0.89	6.75	0.83	6.4
S3	0.45	3.76	0.44	3.53
T1	0.4	3.98	0.32	2.8
T2	0.32	2.74	0.26	1.96
T3	0.28	1.97	0.22	1.36

Tabella 24: Misurazioni di  $I/C$  su tutti gli insiemi di frasi

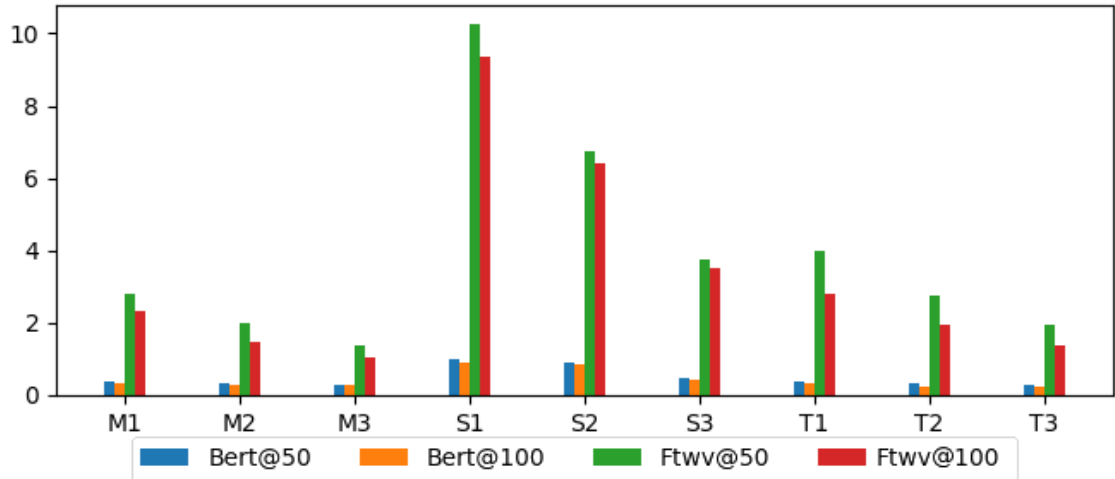


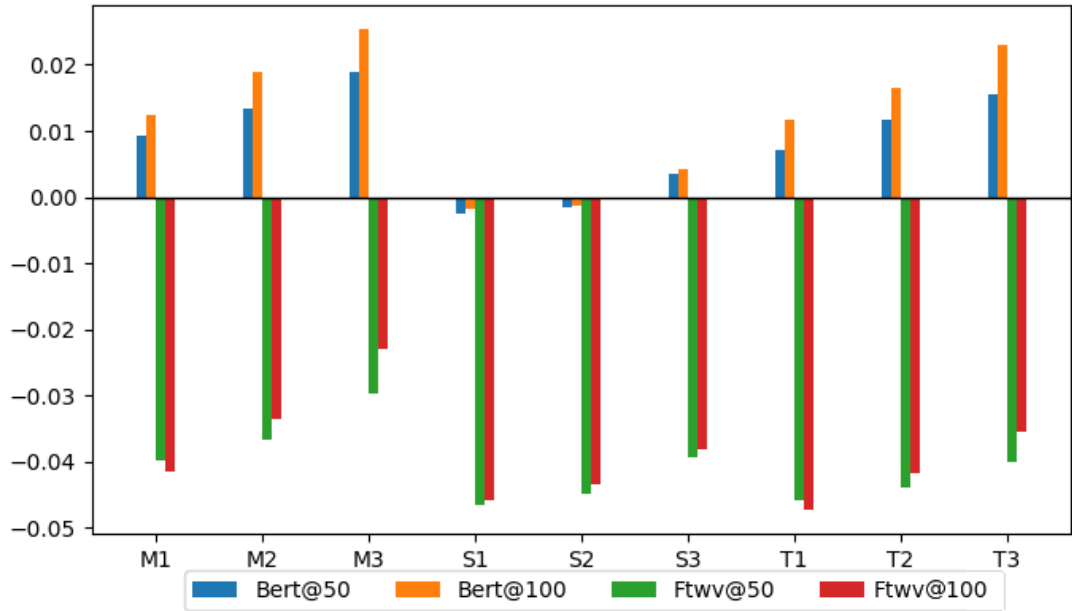
Figura 15: Misurazioni di  $I/C$  su tutti gli insiemi di frasi

**Riduzione nella distanza di Levenshtein ( $LDR$ )** In Tabella 25 e Figura 16 sono riportate le misurazioni per la metrica  $LDR$ . Dati i risultati si possono fare le seguenti osservazioni:

- È possibile confermare quanto visto precedenza: il sistema Bert corregge un maggior numero di errori e ne introduce di meno rispetto al sistema Ftwv. Ciò si traduce in una riduzione positiva della distanza di Levenshtein.
- Anche in questo caso è presente un delta rilevante fra le prestazioni di entrambi i sistemi con frasi di lunghezza diversa. In questo caso, però, la differenza è più marcata nei risultati del sistema bert.
- Il sistema Bert presenta una riduzione negativa sulle pipeline S1 e S2, mentre la riduzione positiva è quasi nulla nella pipeline S3. Questo conferma quanto già osservato, ovvero che il sistema di correzione sviluppato non corregge in modo ottimale i word segmentation error.



Superpipeline	Bert@50	Ftwv@50	Bert@100	Ftwv@100
M1	0.01	-0.04	0.01	-0.04
M2	0.01	-0.04	0.02	-0.03
M3	0.02	-0.03	0.03	-0.02
S1	-0.0	-0.05	-0.0	-0.05
S2	-0.0	-0.04	-0.0	-0.04
S3	0.0	-0.04	0.0	-0.04
T1	0.01	-0.05	0.01	-0.05
T2	0.01	-0.04	0.02	-0.04
T3	0.02	-0.04	0.02	-0.04

Tabella 25: Misurazioni di  $LDR$  su tutti gli insiemi di frasiFigura 16: Misurazioni di  $LDR$  su tutti gli insiemi di frasi

**Distanza di Levenshtein totale ( $LDT$ )** In Tabella 26 e Figura 17 sono riportate le misurazioni per la metrica  $LDT$ . I risultati ottenuti confermano le osservazioni riportate in precedenza.

Superpipeline	Bert@50	Ftwv@50	Bert@100	Ftwv@100
M1	0.06	0.1	0.05	0.11
M2	0.07	0.12	0.07	0.12
M3	0.12	0.16	0.11	0.16
S1	0.02	0.06	0.02	0.06
S2	0.02	0.07	0.02	0.07
S3	0.04	0.08	0.04	0.08
T1	0.04	0.1	0.04	0.1
T2	0.05	0.11	0.05	0.11
T3	0.08	0.13	0.07	0.13

Tabella 26: Misurazioni di  $LDT$  su tutti gli insiemi di frasi

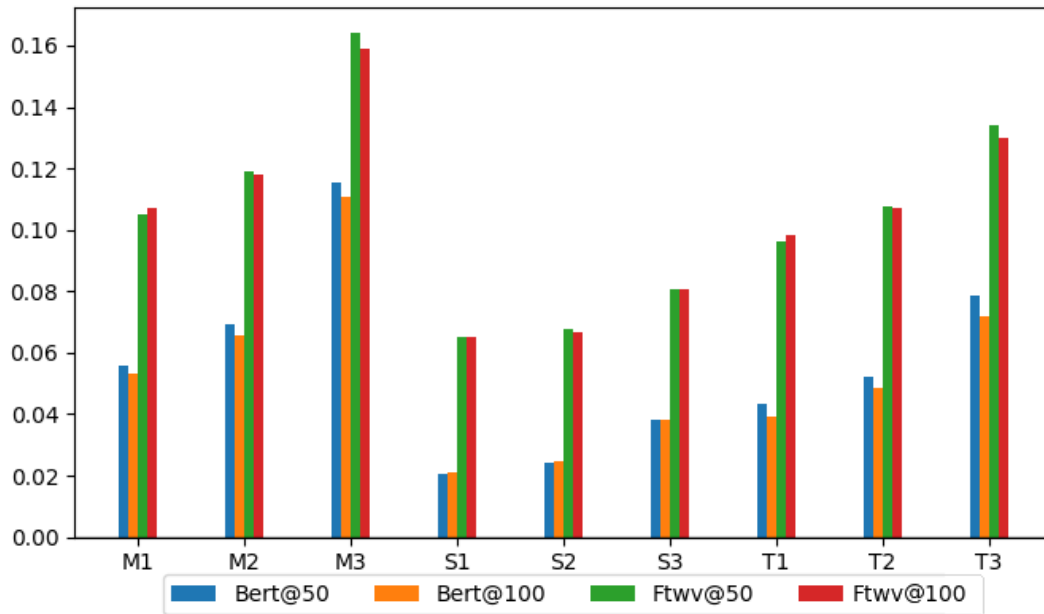


Figura 17: Misurazioni di  $LDT$  su tutti gli insiemi di frasi

**Conclusioni** Dai risultati ottenuti in questa sezione è possibile trarre le seguenti conclusioni:

- Il sistema Bert ottiene performance migliori rispetto al sistema Ftwv per ogni singola metrica. I risultati mostrano come il sistema Bert corregga più errori rispetto al sistema Ftwv. La differenza più marcata risiede però nell'introduzione di nuovi errori, ambito nel quale il sistema Bert ha prestazioni decisamente migliori.
- Una lunghezza maggiore della frase da correggere è correlata ad un aumento delle performance di correzione. Ciò è imputabile al funzionamento dei sistemi di correzione testati, che usano il contesto intorno ad un errore per effettuare la correzione. Più ampio è il contesto, migliori sono i risultati.
- Il sistema Bert ottiene scarsi risultati sulle superpipeline di segmentazione. Ciò è dovuto principalmente al fatto che la correzione è più focalizzata sui word error. La correzione e l'individuazione degli errori di segmentazione, inoltre, è un problema complicato e diverso dalla correzione dei word error: è quindi necessario sviluppare un sistema che riesca ad individuare e correggere più tipi di errori di segmentazione in modo più comprensivo.



# Capitolo 5

## Analisi dell'errore

In questo capitolo sono analizzate le componenti del processo di correzione, con lo scopo di capire quali sono i limiti del sistema e quali siano le parti che più causano un collo di bottiglia alle performance di correzione.

Nella Sezione 5.1 sono date le motivazioni che hanno portato allo svolgimento di questa analisi, della quale è illustrata rapidamente la metodologia. Nella Sezione 5.2 è descritto il processo di modifica del dataset esistente per renderlo consono all'esecuzione dell'analisi. Nelle sezioni 5.3 e 5.4 sono riportate le metriche utilizzate per l'analisi e i risultati ottenuti.

### 5.1 Introduzione

Nel Capitolo 4 sono esposti i risultati dei test sul sistema di correzione. I test svolti valutano le performance complessive del sistema, ma non danno alcuna indicazione rispetto alle fasi che lo compongono. Identificare quali sono le fasi che più influenzano negativamente o positivamente le performance di correzione può essere utile per capire quali sono le componenti da migliorare con più priorità per lo sviluppo del sistema.

L'analisi presentata in questo capitolo riguarda nello specifico la correzione dei word error (Sezione 3.1), e tratta quindi del modulo di correzione token (sotto-sezione 3.2.3). Come visto in precedenza, il modulo di correzione token opera in due fasi: error detection e error correction. L'analisi, che verte sulla fase di error correction, valuterà:

- La capacità del modello BERT di produrre la soluzione per la correzione fra i vari candidati. La presenza della soluzione fra i candidati è condizione necessaria per poter applicare la giusta correzione: è quindi importante capire qual è il limite superiore che la generazione dei candidati pone al processo di correzione.

- La capacità del sistema di correzione di scegliere il candidato corretto, se presente, fra quelli proposti dal modello BERT. Verrà valutata anche la capacità del sistema di evitare di apportare correzioni nel caso BERT non produca la soluzione fra i candidati.

La valutazione di questi due step del processo di correzione permetterà di stabilire quali sono i margini di miglioramento nella fase di error correction, considerando il limite superiore imposto dalla generazione dei candidati del modello BERT. Questo, confrontato con i risultati dei testi illustrati nel Capitolo 4, permetterà di valutare anche il margine di errore della fase di error detection.

## 5.2 Creazione di un dataset specifico

Come spiegato nell'introduzione, l'analisi presentata in questo capitolo verte unicamente sulla parte di error correction. È quindi chiaro come la metodologia dell'analisi deve prevedere un modo per escludere l'error detection dal processo di correzione. Ciò implica che è necessario disporre di frasi in cui gli errori sono in posizione nota, in modo da non dover fare error detection.

**Metodologia** L'idea è quella di perturbare ulteriormente le frasi presenti in `dst@100`, introducendo un solo word error per frase. Il token perturbato non viene però reinserito nella frase, ma viene mascherato, permettendo così di conoscere la posizione dell'errore da correggere e di avere una frase già pronta per la generazione dei candidati.

Ad esempio, data la frase già perturbata:

*"Sarà n e c e s s a r i o uno sforzo straordinario per mobilitare le risorse."*

si introduce un ulteriore word error, e si maschera la posizione dell'errore, ottenendo la seguente tripletta:

1. Frase: *"Sarà n e c e s s a r i o uno sforzo [MASK] per mobilitare le risorse."*
2. Token originale: *"straordinario"*
3. Token perturbato: *"strnordinnrio"*

Quanto appena spiegato informalmente è un processo divisibile nelle seguenti fasi:

1. Estrazione
2. Perturbazione

**Estrazione** Lo scopo di questa fase è quello di ottenere una lista di frasi perturbate con diverse superpipeline. Nel contesto di questo capitolo si intende una frase come una stringa senza l'aggiunta di ulteriori metadati. Per ogni sample  $s \in dst@100$  è possibile estrarre due tipi di frase:

- La frase originale non perturbata, presente nel campo *text*.
- Una frase perturbata scelta dal campo *perturbed*. Ogni frase perturbata è identificata dalla superpipeline usata per la sua perturbazione.

Si ottiene l'insieme delle frasi estratte  $D_{extr}$  estraendo casualmente frasi da  $dst@100$  in modo da rispettare la distribuzione in Tabella 27:

Tipo frase	Num. frasi
text	3500
T1	3500
T2	3500
T3	3500
S1	3500
S2	3500
S3	3500
M1	3500
M2	3500
M3	3500
<b>Totale</b>	<b>35000</b>

Tabella 27: Distribuzione delle frasi estratte. Per "tipo" si intende *text* se la frase non è perturbata; se invece la frase è perturbata il tipo corrisponde alla superpipeline utilizzata

Ogni frase è inoltre associata al suo tipo: in questo modo durante l'analisi sarà possibile valutare i risultati anche a base dell'intensità della perturbazione.

**Perturbazione** Per ogni frase  $f \in D_{extr}$  il primo passaggio nella fase di perturbazione consiste nel tokenizzare la frase. Si ottiene quindi una lista di token  $T = [t_1, \dots, t_n]$ . Questo passaggio si rende necessario in quanto si vuole introdurre un errore all'interno di un singolo token.

Fra lista di token ottenuta dalla tokenizzazione si individua un subset  $P \subseteq T$  di token detti "perturbabili". Un token  $t$  è considerato perturbabile se soddisfa le seguenti condizioni:

- Dato lo stesso vocabolario  $V$  usato per l'error detection (Sezione 3.2.3),  $t \in V$ .

- $t$  è lungo almeno due caratteri.

Queste condizioni servono ad evitare di ri-perturbare un token che è già stato perturbato in precedenza o che deriva dallo split di un altro token. Si pensi ad esempio a una situazione in cui "papa" viene perturbato come "p a p a": pur essendo "a" presente nel vocabolario, non sarebbe corretto considerare "a" come un token perturbabile.

Si sceglie quindi in modo casuale un solo token  $t_p$  da perturbare dall'insieme  $P$ . Per perturbare  $t_p$  si usa la funzione  $f_{alternative}$  del modulo di characters replacement (sottosezione 2.3.1). Quindi, si ha che:

$$t'_p = f_{alternative}(t_p) \quad (69)$$

Successivamente si sostituisce  $t_p$  nell'insieme  $T$  con il token "[MASK]", ottenendo la lista di token  $T' = [t_1, \dots, [MASK], \dots, t_n]$ . Infine, si detokenizza  $T'$  per ottenere la frase mascherata  $f'$ .

Alla fine di questa fase, per ogni frase  $f \in D_{extr}$  è prodotta una tupla strutturata come in Tabella 28:

Campo	Contenuto
<i>sent</i>	Frase originale mascherata ( $f'$ )
<i>maskedTok</i>	Token che è stato mascherato ( $t'_p$ )
<i>correctTok</i>	Token originale non perturbato ( $t_p$ )
<i>pertSup</i>	Tipo di $f$ (Tabella 27)

Tabella 28: Tupla prodotta per ogni frase  $f$  dalla fase di perturbazione

L'insieme delle tuple risultanti ottenuto da questa fase è detto  $D_{an}$ .

**Esempio** Si prenda ad esempio la frase fornita come esempio all'inizio di questa sezione, perturbata con la superpipeline S1:

*"Sarà n e c e s s a r i o uno sforzo straordinario per mobilitare le risorse."*

La fase di perturbazione ha inizio tokenizzando la frase:

'Sarà', 'n', 'e', 'c', 'e', 's', 's', 'a', 'r', 'i', 'o', 'uno', 'sforzo', 'straordinario',  
'per', 'mobilitare', 'le', 'risorse', '.'

Si può notare come siano stati sottolineati tutti i token perturbabili. Viene scelto casualmente il token 'straordinario'. Si produce quindi una versione perturbata del token scelto:

$$f_{alternative}('straordinario') = 'strnordinnrio' \quad (70)$$



Si procede sostituendo il token perturbato con *[MASK]* all'interno della lista dei token:

*'Sarà', 'n', 'e', 'c', 'e', 's', 's', 'a', 'r', 'i', 'o', 'uno', 'sforzo', '[MASK]', 'per',  
'mobilitare', 'le', 'risorse', '.'*

Si detokenizza in seguito la lista di token:

*"Sarà n e c e s s a r i o uno sforzo [MASK] per mobilitare le risorse."*

Si produce quindi la tupla in Tabella 29:

Campo	Contenuto
<i>sent</i>	<i>"Sarà n e c e s s a r i o uno sforzo [MASK] per mobilitare le risorse."</i>
<i>maskedTok</i>	<i>"strnordinnrio"</i>
<i>correctTok</i>	<i>"straordinario"</i>
<i>pertSup</i>	S1

Tabella 29: Tupla prodotta per ogni frase  $f$  dalla fase di perturbazione

### 5.3 Analisi generazione dei candidati

La generazione dei candidati è la prima parte dello stadio di candidates generation and picking della fase di error correction del modulo di correzione token (sottosezione 3.2.3). La generazione dei candidati sfrutta un modello BERT allenato sul masked language modelling (sottosezione 3.2.2), per produrre i possibili candidati al rimpiazzo del token *[MASK]* in una frase data in input. Si ricorda inoltre che ogni candidato prodotto da BERT è associato alla propria probabilità di essere il rimpiazzo corretto.

**Metriche** L'analisi della generazione dei candidati mira quindi a quantificare la capacità del modello BERT di produrre la soluzione fra i candidati proposti. Si intende inoltre misurare quanto accurata sia la probabilità che il modello associa ad ogni candidato. Per fare ciò verranno misurati:

- **S@k** (soluzione entro k): dato un campione di  $n$  frasi, indica la percentuale di volte un cui la soluzione è presente nei primi  $k$  candidati prodotti da BERT (si ricorda che i candidati sono ordinati per probabilità).

$$S@k = \frac{\# \text{ frasi con soluzione nei top-}k \text{ risultati}}{n} \quad (71)$$

Questa metrica è misurata per i seguenti valori di  $k$ : 10, 20, 30.

- **Distribuzione soluzioni:** per le frasi in cui la soluzione è prodotta nei primi  $k$  candidati, è la distribuzione che tiene traccia della posizione della soluzione nell'ordinamento per probabilità dei candidati. La distribuzione è calcolata per un valore di  $k$  corrispondente a 30.

**Risultati** Si riportano in Tabella 30 e Figura 18 i risultati ottenuti dalla misurazione di  $S@k$ .

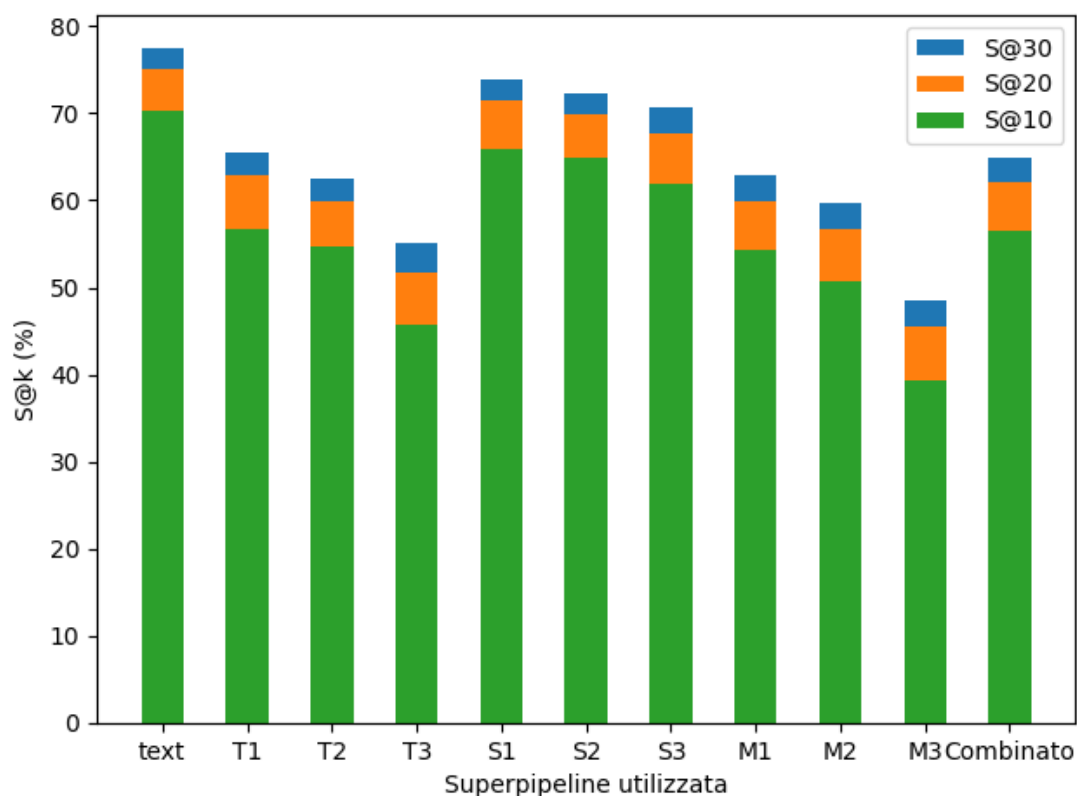


Figura 18:  $S@k$  misurato su ogni tipo di frase. Combinato indica  $S@k$  misurato su tutto il dataset

<b>Tipo frase</b>	<b>S@10</b>	<b>S@20</b>	<b>S@30</b>
text	70.23	74.98	77.41
T1	56.65	62.87	65.42
T2	54.76	59.98	62.47
T3	45.78	51.75	55.03
S1	65.88	71.46	73.77
S2	64.98	69.9	72.24
S3	61.96	67.59	70.62
M1	54.33	59.82	62.94
M2	50.62	56.63	59.76
M3	39.37	45.52	48.51
Combinato	56.47	62.07	64.83

Tabella 30: S@k misurato su ogni tipo di frase (risultati riportati in percentuale). Combinato indica S@k misurato su tutto il dataset

Da questi risultati si possono trarre alcune considerazioni:

- Come era facilmente prevedibile, le superpipeline più perturbate ottengono risultati visibilmente peggiori proporzionalmente al livello di perturbazione. Ciò è dato dal fatto che BERT fa uso del contesto intorno a [MASK] per provare a indovinare la parola originale. Se il contesto è sporcato, è lecito aspettarsi prestazioni più basse.
- La generazione dei candidati risente meno dei segmentation error rispetto ai word error. Ciò potrebbe essere dovuto all'algoritmo di segmentazione Word-Piece usato da BERT, che, dividendo una frase in subtoken (parti più piccole di token), potrebbe risultare particolarmente resiliente ai segmentation error. Al momento però questa è solo un'ipotesi che andrebbe approfondita e verificata.
- Ad un maggiore valore di  $k$  corrisponde naturalmente anche un incremento di  $S@k$ . Se da un lato questo significa che la soluzione è teoricamente presente in più casi, e quindi si ha un limite superiore più alto per le performance di correzione, dall'altro avere più candidati può portare a introdurre più errori durante la fase di scelta dei candidati. È quindi necessario scegliere un valore di  $k$  sufficientemente alto da produrre la soluzione, ma non così alto da far sì che si confonda con gli altri candidati. Valutando la differenza fra  $S@10$  e  $S@20$ , che è di 5,6 punti percentuali sull'intero dataset, si può concludere che l'incremento sia decisamente significativo. La differenza  $S@20$  e  $S@30$  corrisponde invece a 2,7 punti percentuali. Alla luce di questa osservazione, il sistema di correzione implementato utilizza un parametro  $k$  pari a 20.

Quanto appena detto si può notare nel grafico della distribuzione delle soluzioni (Figura 19). Dalla distribuzione si nota come la maggior parte delle soluzioni rientri nei primi candidati, e che la presenza della soluzione al candidato  $n$ -esimo diminuisca esponenzialmente man mano che ci si allontana dalla prima posizione.

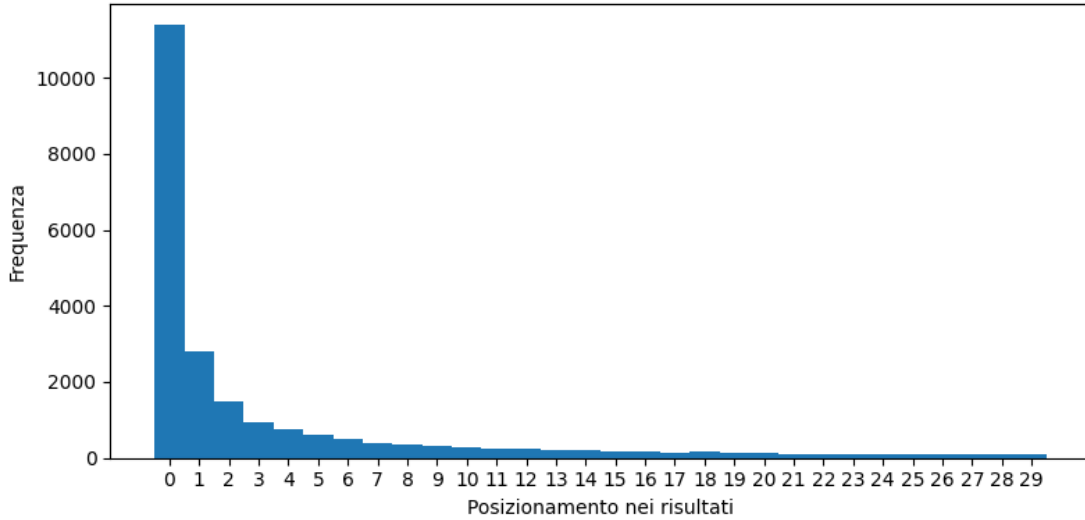


Figura 19: Distribuzione dei risultati in base all'ordinamento per probabilità prodotto da BERT di tutte le frasi del dataset (combinato). Non sono riportate le distribuzioni degli altri dataset per brevità, dato che sono pressochè sovrapponibili a quella in figura.

## 5.4 Analisi scelta dei candidati

La scelta dei candidati comprende la seconda parte dello stadio candidates generation and picking e lo stadio di validation (sottosezione 3.2.3). Data la lista dei  $k$  candidati  $C = [c_1, \dots, c_k]$  prodotti da BERT per una frase  $f_{mask}$  nella quale la parola  $s$  è mascherata, la scelta dei candidati deve:

1. Scegliere il candidato corretto se  $s \in C$ , ovvero se la soluzione è presente fra i candidati.
2. Non apportare alcuna correzione se  $s \notin C$ , ovvero se la soluzione non è presente fra i candidati.

**Metriche** Sulla base di quanto appena esposto, sono definite alcune metriche per analizzare il comportamento del sistema di correzione. Le metriche definite si dividono in due categorie:

- Metriche misurate nei casi in cui  $s \in C$ .
- Metriche misurate nei casi in cui  $s \notin C$ .

Nel caso in cui  $s \in C$  sono definite le seguenti metriche:

- **Selezioni corrette:** percentuale dei casi in cui il sistema di correzione sceglie un candidato  $c \in C$ , e  $c$  coincide con  $s$ . Ovvero, è la percentuale dei casi in cui il sistema seleziona un candidato, e il candidato selezionato è quello corretto.
- **Selezioni errate:** percentuale dei casi in cui il sistema di correzione sceglie un candidato  $c \in C$ , e  $c$  non coincide con  $s$ . Ovvero, è la percentuale dei casi in cui il sistema seleziona un candidato, e il candidato selezionato non è quello corretto.
- **Selezioni mancate:** percentuale dei casi in cui il sistema di correzione non sceglie alcun candidato, ignorando quindi la presenza della soluzione in  $C$ .

Nel caso in cui  $s \notin C$  sono definite le seguenti metriche:

- **Non-selezioni corrette:** percentuale dei casi in cui il sistema di correzione non sceglie alcun candidato, evitando così di introdurre un errore.
- **Non-selezioni mancate:** percentuale dei casi in cui il sistema di correzione sceglie un candidato  $c \in C$ , introducendo così un errore.

Possiamo classificare le metriche appena elencate in due categorie distinte:

- **Scelte Corrette:** sono tutte le scelte che il sistema prende che hanno un impatto positivo sulla correzione. Sono date dalle *Selezioni corrette* e dalle *Non-selezioni corrette*.
- **Scelte Errate:** sono tutte le scelte che il sistema prende che hanno un impatto negativo sulla correzione. Sono date dalle *Selezioni errate*, *Selezioni mancate* e dalle *Non-selezioni mancate*.

**Risultati** In Tabella 31 sono riportate le misurazioni per le metriche in cui la soluzione è presente fra i candidati prodotti.

Anche in questo caso, come è lecito aspettarsi, le frasi perturbate con superpeline che introducono più rumore sono quelle che ottengono risultati peggiori, con una differenza fra i risultati migliori e peggiori contenuta entro l'8% per quanto riguarda le selezioni corrette. Nella discussione dei risultati si farà riferimento solo alla combinazione di tutti i tipi di frase per brevità. Tutte le considerazioni sono

comunque applicabili anche per tutti gli altri tipi di frase, per i quali i risultati differiscono solo sensibilmente.

Come visibile nel grafico in Figura 20, l'euristica di selezione del candidato produce la scelta corretta nell'85,3% dei casi. Come mostrato in Figura 21, infatti, circa l'89% delle soluzioni si trova nella prima posizione fra i candidati ordinati per la distanza di Levenshtein. Si ricorda che l'euristica illustrata in Sezione 3.2.3 sceglie sempre il candidato in prima posizione per ordinamento secondo la distanza di Levenshtein. Da questo è possibile analizzare in modo più critico i risultati ottenuti: l'euristica identifica la soluzione nell'89% dei casi, ma alcune soluzioni (poco più del 3%) sono scartate dallo stadio di validazione. Il resto della percentuale di errore è dovuto all'11% di soluzioni che si trovano dalla seconda posizione in poi nell'ordinamento per distanza di Levenshtein.

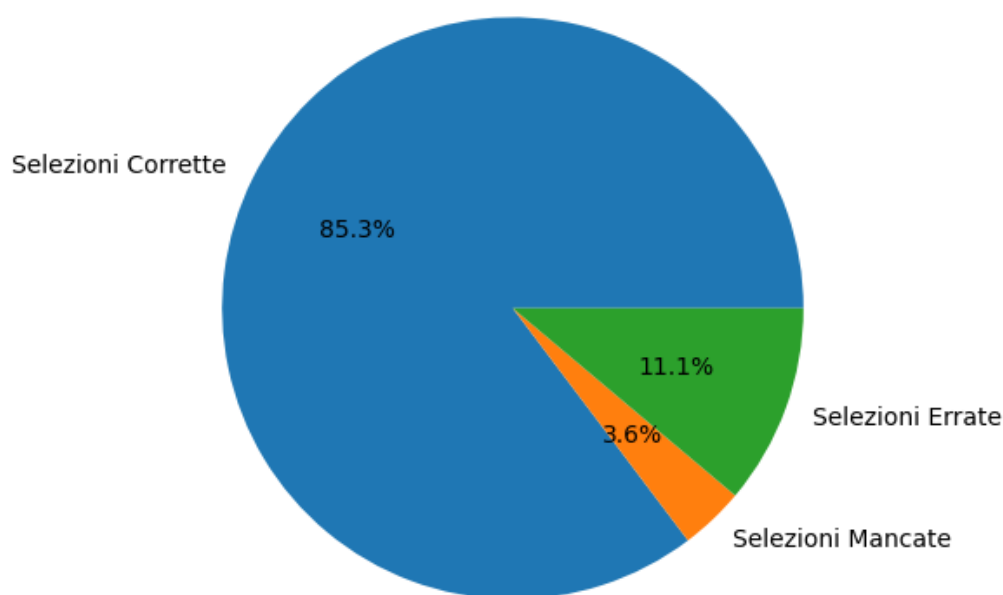


Figura 20: Metriche misurate per frasi in cui  $s \in C$  sull'intero dataset

Pert.	Sel. Corrette	Sel. Errate	Sel. mancate
text	88.16	8.8	3.05
T1	86.41	10.0	3.59
T2	83.97	12.02	4.01
T3	80.17	16.69	3.14
S1	86.93	9.15	3.93
S2	87.85	9.31	2.84
S3	86.41	9.53	4.06
M1	85.03	11.2	3.77
M2	84.07	12.05	3.88
M3	80.29	16.12	3.58
Combinato	85.3	11.13	3.58

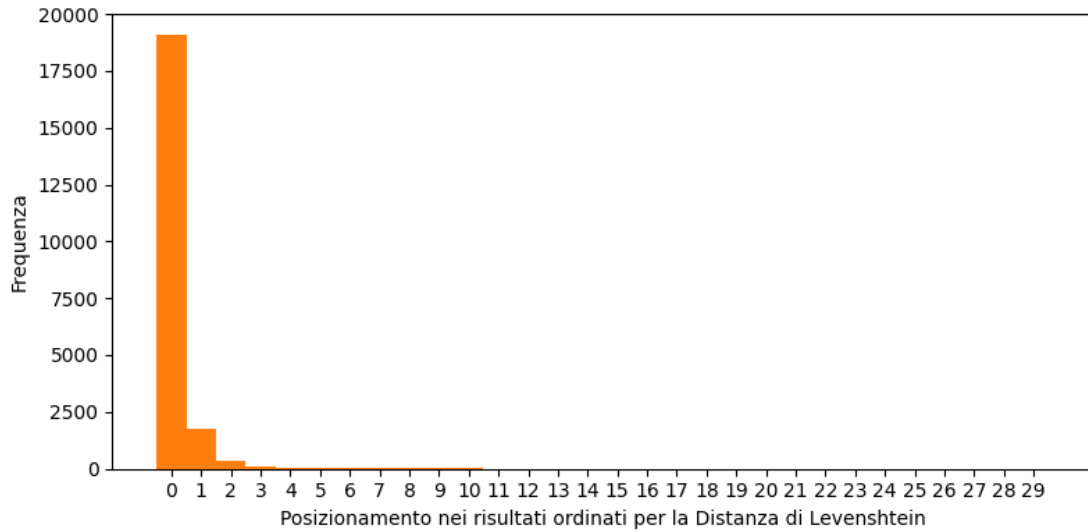
Tabella 31: Metriche misurate per frasi in cui  $s \in C$ 

Figura 21: Distribuzione della posizione della soluzione all'interno dei risultati ordinati per la distanza di Levenshtein

Si analizzano ora i casi in cui la soluzione non è presente fra candidati. In Tabella 32 e in sono riportate le statistiche per questo tipo di frasi.

<b>Pert.</b>	<b>N.Sel. Corrette</b>	<b>N.Sel. Mancate</b>
text	72.72	27.28
T1	67.71	32.29
T2	67.8	32.2
T3	64.84	35.16
S1	72.75	27.25
S2	71.96	28.04
S3	70.14	29.86
M1	68.37	31.63
M2	67.47	32.53
M3	59.65	40.35
Combinato	67.5	32.5

Tabella 32: Metriche misurate per frasi in cui  $s \notin C$

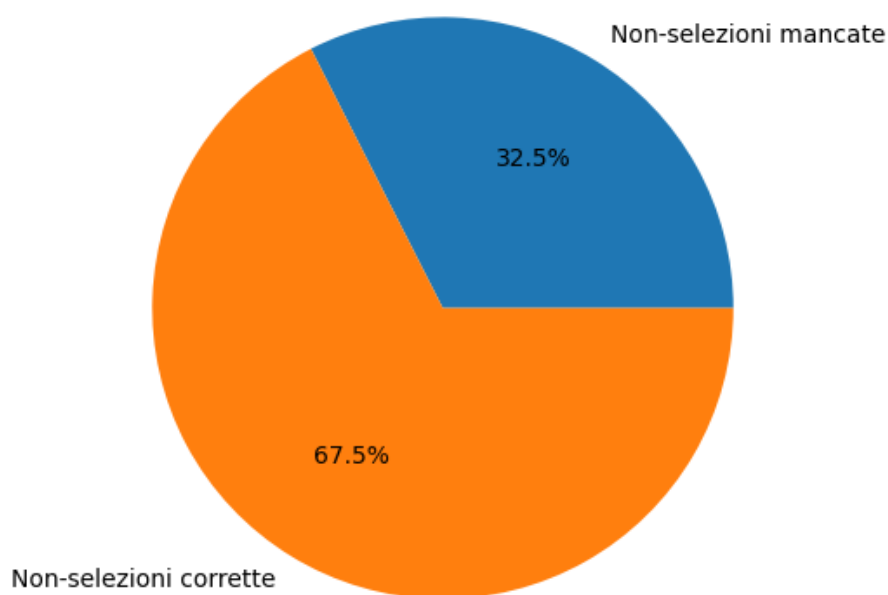


Figura 22: Metriche misurate per frasi in cui  $s \notin C$  sull'intero dataset



Seppur la maggior parte delle scelte che il sistema prende in questo caso sia corretta, è presente in ogni caso un'elevata percentuale di non-selezioni mancate. Si ricorda inoltre che per ogni errore viene sempre scelto un candidato per la correzione. È compito dello stadio di validazione filtrare eventuali correzioni errate. La percentuale di non-selezioni errate è quindi da attribuire unicamente a questo stadio. Nell'eseguire tale analisi bisogna però anche tenere conto che nella maggioranza dei casi BERT riesce a produrre la soluzione fra i candidati (62,07% dei casi, da Tabella 30), motivo per cui è ragionevole cercare di mantenere un certo bias verso questi casi.

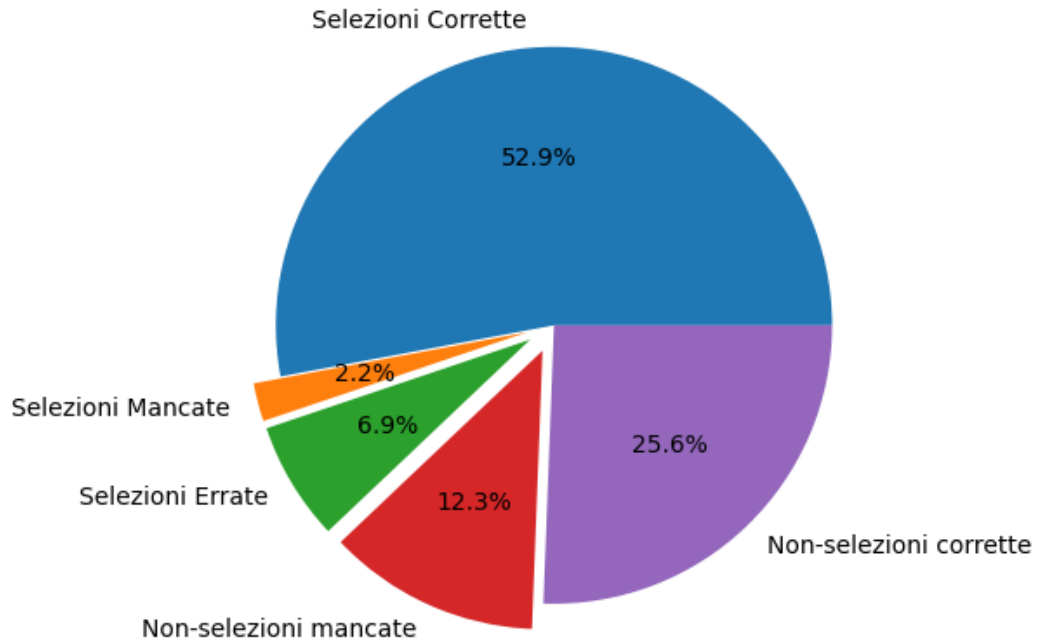


Figura 23: Metriche misurate sia per frasi in cui  $s \notin C$ , sia per frasi in cui  $s \in C$  sull'intero dataset. Le fette esplose sono le Scelte Corrette, le rimanenti sono le Scelte Errate.

Per concludere questa analisi, in Figura 23 sono illustrate tutte le metriche discusse in questo capitolo. Nel comporre il grafico, si è tenuto conto della diversa cardinalità degli insiemi di frasi in cui  $s \in C$  e  $s \notin C$ : il peso di ogni metrica è quindi proporzionale al numero di frasi su cui è stata misurata. Si può quindi

notare come il sistema di correzione effettui una scelta corretta nel 78,55% dei casi, e prenda una scelta sbagliata nel 21,45% dei casi (in Tabella 33 questa statistica è riportata per ogni tipo di frase).

<b>Tipo Frase</b>	<b>Scelte Corrette</b>	<b>Scelte Errate</b>
text	84.29	15.71
T1	79.47	20.53
T2	77.5	22.5
T3	72.77	27.23
S1	82.88	17.12
S2	83.06	16.94
S3	81.14	18.86
M1	78.34	21.66
M2	76.87	23.13
M3	69.04	30.96
Combinato	78.55	21.45

Tabella 33: Percentuale di scelte positive e negative per tutti i tipi di frase

# Conclusioni

In questa tesi è stato sviluppato e proposto un approccio automatico al problema dell'OCR post processing, basato sull'uso di un modello BERT pre-allenato. Il sistema sviluppato è strutturato come una pipeline, nella quale ogni modulo corregge un particolare tipo di errore. Sono stati sviluppati due moduli, denominati modulo di correzione Token e modulo di correzione Split per correggere rispettivamente i non-word error e gli space splitting error.

Sono state definite una metodologia di test e delle metriche apposite per valutare le performance del sistema sviluppato. Le metriche definite mirano a valutare il numero di errori corretti e introdotti dal sistema di correzione. La stessa metodologia di test è stata applicata ad un approccio presente in letteratura, in modo tale da poter comparare i risultati ottenuti.

Data la difficoltà nel trovare un dataset con sequenze parallele di testo acquisito tramite OCR e testo corretto, è stato sviluppato un sistema per introdurre in maniera controllata rumore in sequenze di testo pulite. Ciò ha consentito di ottenere un dataset su cui è stato possibile eseguire i test stabiliti. Controllare l'introduzione del rumore ha inoltre reso possibile definire vari livelli di intensità di perturbazione, rendendo possibile lo studio delle performance del sistema di OCR post processing sviluppato al variare del tasso di errore.

I risultati ottenuti mostrano come il sistema sviluppato, a seconda del livello di rumore presente all'interno del testo, riesca a correggere fra il 25% e il 40% degli errori presenti, a fronte di una minima quantità di errori introdotti.

È stata infine eseguita un'analisi sulle prestazioni del sistema, che ha permesso di individuarne le maggiori criticità per eventuali sviluppi futuri. Il lavoro svolto in questa tesi dimostra come l'utilizzo di un modello BERT pre-allenato sia un approccio percorribile per il problema dell'OCR post-processing, sebbene l'analisi dell'errore dimostri come il sistema sviluppato abbia ampi margini di miglioramento.

**Sviluppi futuri** I risultati dei test e l'analisi dell'errore nei capitoli 4 e 5 fanno emergere alcune aree in cui è possibile migliorare il sistema sviluppato:

- I test hanno dimostrato come all'aumentare della lunghezza del testo in input il sistema produca risultati migliori. Potrebbe quindi essere opportuno usare frasi di 256 caratteri (limite superiore per BERT) e portare avanti la correzione con un approccio a sliding window sul testo. Questo approccio avrebbe il vantaggio di massimizzare la lunghezza delle frasi e di eseguire automaticamente più iterazioni su errori non corretti. Sarebbe però necessario un nuovo metodo per valutare le prestazioni del sistema su segmenti di testo più lunghi.
- L'analisi dell'errore ha dimostrato come la parte di error correction del sistema agisca correttamente almeno nel 70% dei casi (nel caso del tipo di frase con performance peggiori). Il sistema di correzione, nel migliore dei casi, corregge il 40% degli errori presenti. Buona parte di questo scarto può essere attribuibile alla fase di error detection, che andrebbe riscritta rimpiazzando l'attuale sistema che fa semplicemente uso di un vocabolario.
- I test fanno emergere come il sistema di correzione non abbia ottime performance sui word segmentation error. Sarebbe quindi opportuno esplorare altre soluzioni per questo tipo di errori. In questo ambito, l'analisi della letteratura suggerisce come approcci basati su NMT possano ottenere buoni risultati (sottosezione 1.1.2).
- Durante l'affinamento del sistema di correzione, si è provato a eseguire il fine-tuning del modello BERT pre-allenato, ri-allenando il modello su una parte del dataset. I test, eseguiti sulla parte rimanente del dataset, hanno prodotto al più risultati lievemente inferiori a quelli del modello pre-allenato. Sarebbe quindi interessante provare a rivedere questo approccio, cercando sia di espandere il dataset ottenuto, sia di allungare le frasi al suo interno per cercare di ottenere risultati migliori.
- Nel modulo di correzione token, durante la scelta dei candidati, si scarta il valore di probabilità prodotto da BERT in favore della scelta del candidato con la minor distanza di Levenshtein. Si potrebbe invece utilizzare il valore di probabilità in congiunzione con la distanza di Levenshtein per sviluppare dei metodi di scelta e validazione più accurati.

# Bibliografia

- [1] Adam Jatowt, Mickael Coustaty, Nhu-Van Nguyen, Antoine Doucet, et al. Deep statistical analysis of ocr errors for effective post-ocr processing. In *2019 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, pages 29–38. IEEE, 2019.
- [2] Roger T Hartley and Kathleen Crumpton. Quality of ocr for degraded text images. *arXiv preprint cs/9902009*, 1999.
- [3] Guillaume Chiron, Antoine Doucet, Mickaël Coustaty, Muriel Visani, and Jean-Philippe Moreux. Impact of ocr errors on the use of digital libraries: towards a better access to information. In *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, pages 1–4. IEEE, 2017.
- [4] Stephen Mutuvi, Antoine Doucet, Moses Odeo, and Adam Jatowt. Evaluating the impact of ocr errors on topic modeling. In *International Conference on Asian Digital Libraries*, pages 3–14. Springer, 2018.
- [5] Rose Holley. How good can it get? analysing and improving ocr accuracy in large scale historic newspaper digitisation programs. *D-Lib Magazine*, 15(3/4), 2009.
- [6] Wojciech Bieniecki, Szymon Grabowski, and Wojciech Rozenberg. Image preprocessing for improving ocr accuracy. In *2007 international conference on perspective technologies and methods in MEMS design*, pages 75–80. IEEE, 2007.
- [7] Yaakov HaCohen-Kerner. What is difference between string and token in natural language processing techniques?, 06 2014.
- [8] Wikipedia. N-gramma — wikipedia, l’enciclopedia libera, 2020. [Online; in data 3-novembre-2021].
- [9] Thorsten Brants and Alex Franz. Web 1t 5-gram version 1, set 2006.

- [10] Youssef Bassil and Mohammad Alwani. Ocr context-sensitive error correction based on google web 1t 5-gram data set. *arXiv preprint arXiv:1204.0188*, 2012.
- [11] Andrew Carlson and Ian Fette. Memory-based context-sensitive spelling correction at web scale. In *Sixth International Conference on Machine Learning and Applications (ICMLA 2007)*, pages 166–171. IEEE, 2007.
- [12] Kevin Atkinson. Gnu aspell, 1998. *Software available at <http://aspell.net>*.
- [13] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.
- [14] Kareem Mokhtar, Syed Saqib Bukhari, and Andreas Dengel. Ocr error correction: State-of-the-art vs an nmt-based approach. In *2018 13th IAPR International Workshop on Document Analysis Systems (DAS)*, pages 429–434. IEEE, 2018.
- [15] Mika Härmäläinen and Simon Hengchen. From the past to the future: a fully automatic nmt and word embeddings method for ocr post-correction. *arXiv preprint arXiv:1910.05535*, 2019.
- [16] Thi Tuyet Hai Nguyen, Adam Jatowt, Nhu-Van Nguyen, Mickael Coustaty, and Antoine Doucet. Neural machine translation with bert for post-ocr error detection and correction. In *Proceedings of the ACM/IEEE Joint Conference on Digital Libraries in 2020*, pages 333–336, 2020.
- [17] Vivi Nastase and Julian Hitschler. Correction of ocr word segmentation errors in articles from the acl collection through neural machine translation methods. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.
- [18] Quan Duong, Mika Härmäläinen, and Simon Hengchen. An unsupervised method for ocr post-correction and spelling normalisation for finnish. *arXiv preprint arXiv:2011.03502*, 2020.
- [19] Chantal Amrhein and Simon Clematide. Supervised ocr error detection and correction using statistical and neural machine translation methods. *Journal for Language Technology and Computational Linguistics (JLCL)*, 33(1):49–76, 2018.

- [20] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M Rush. Opennmt: Open-source toolkit for neural machine translation. *arXiv preprint arXiv:1701.02810*, 2017.
- [21] Rico Sennrich, Orhan Firat, Kyunghyun Cho, Alexandra Birch, Barry Haddow, Julian Hitschler, Marcin Junczys-Dowmunt, Samuel Läubli, Antonio Valerio Miceli Barone, Jozef Mokry, et al. Nematus: a toolkit for neural machine translation. *arXiv preprint arXiv:1703.04357*, 2017.
- [22] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [23] Mahdi Hajiali, Jorge Ramón Fonseca Cacho, and Kazem Taghva. Generating correction candidates for ocr errors using bert language model and fasttext subword embeddings. In *Intelligent Computing*, pages 1045–1053. Springer, 2022.
- [24] Shaohua Zhang, Haoran Huang, Jicong Liu, and Hang Li. Spelling error correction with soft-masked bert. *arXiv preprint arXiv:2005.07421*, 2020.
- [25] Edward Loper and Steven Bird. Nltk: The natural language toolkit. *arXiv preprint cs/0205028*, 2002.