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

dedicato a . . .

Prefazione

hkjafgyruet.

Organizzazione della tesi

La tesi è organizzata come segue:

- nel Capitolo 1 ....

Ringraziamenti

asdjhgftry.

Indice

Capitolo 1

Introduzione

Capitolo 2

Stato dell'Arte

In questo capitolo si passa in rassegna lo stato dell'arte riguardante il tema dell'OCR post-processing.
Nella sezione 2.1 è presentata un' introduzione al problema affrontato e alcune definizioni di base. Nella sezione 2.2 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.

2.1 Introduzione

Optical Character Recognition 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 risultati 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).

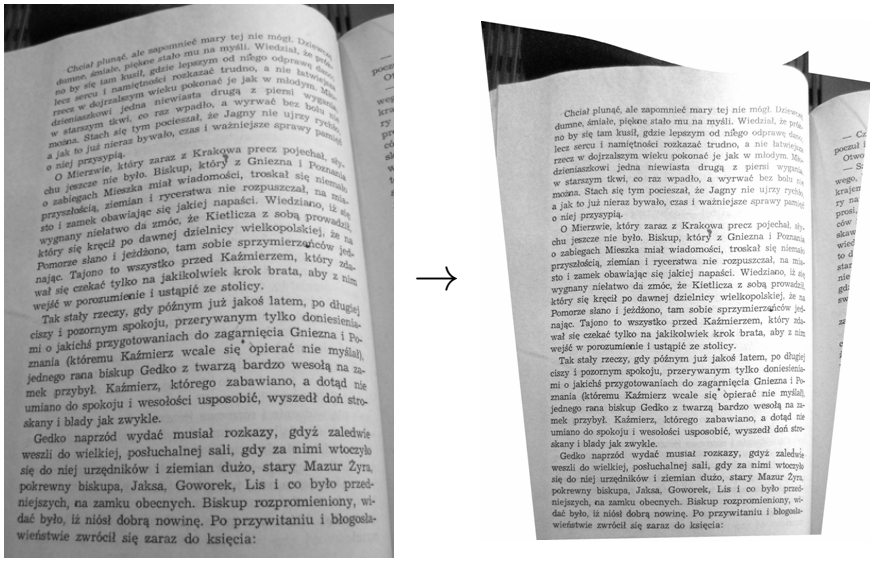


Figura 1: A sinistra foto di una pagina contenente del testo. A destra, foto della stessa pagina pre-processata per facilitare l'estrazione del testo. Esempio preso da [6].

- **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 2.2.

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

2.2 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 per letteratura, raggruppati nelle seguenti categorie di metodologie:

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

2.2.1 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:

$$Tok : F \to T \tag{1}$$

dove ogni $t \in T$ è una lista $[t_1, ..., t_n]$ 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. Dato 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 amplio 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, ma 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.

2.2.2 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 dell 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).

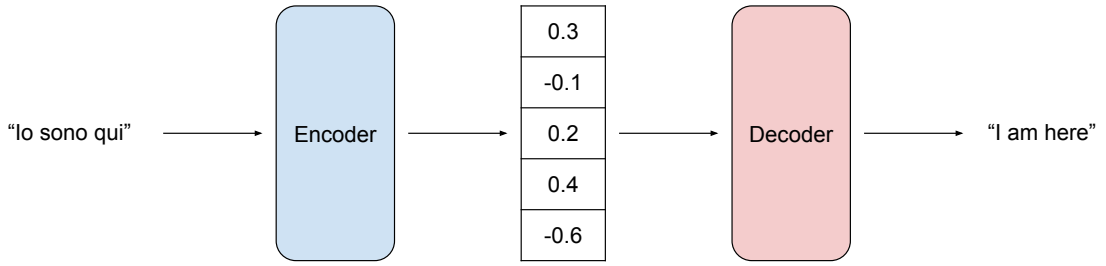


Figura 2: Schema semplificato di un'architettura encoder-decoder che traduce da italiano a inglese

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:

$$M : E \rightarrow C \tag{2}$$

Allo stesso modo, ogni modello necessita prima di essere allenato con corpus parallelo dove ad ogni sequenza $e \in E$ corrisponde un 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 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 un avere un modello con troppo bias verso le sequenze senza errori. L'approccio per la correzione si divide dunque in due fasi: un prima fase di error detection utilizza un modello BERT (si rimanda il lettore alla sottosezione 2.2.3) 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 4.1), 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.

2.2.3 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 desta 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 4.2.2) 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 3

Dataset e Perturbazione

In questo capitolo è descritto il processo di creazione del dataset usato per valutare il sistema di correzione sviluppato.
Nella sezione 3.1 sono discussi gli obiettivi e le caratteristiche che il dataset deve avere. Nella sezione 3.2 sono delineate la metodologia di creazione e la struttura del dataset. Nella sezione 3.3 è descritto il processo introduzione di errori all'interno delle frasi per simularne l'acquisizione tramite OCR, detto perturbazione. Nella sezione 3.4 infine, sono discussi i parametri e la configurazione del dataset.

3.1 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 in TODO: inserire cit a cap 5. 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.

3.2 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`. TODO: Domanda: Devo citare il lab per il dataset? Se si, come? 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 trasformalo 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 in TODO: ref a capitolo 5, 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 non presenti fra 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:

$$D_{fl} = \{d \; \forall d \in D \mid language_d \in L\} \tag{3}$$

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 3.4.

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:

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

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}

$$D_{es} = \bigcup_{p \in D_{es1}} p \tag{5}$$

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 dalla altre frasi estratte dal medesimo paragrafo.
- Ogni frase è compresa fra una lunghezza minima l_{min} e una lunhezza massima l_{max} (sezione 3.4).

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 riutilizza 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 riutilizza 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 riutilizza come input della procedura di frammentazione, tornando al punto 1.

La procedura appena descritta fa in modo che i frammenti approssimino il più possibile 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 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 frase di frammentazione si può quindi formalizzare in due step. Nel primo step, ogni paragrafo viene frammentato nella lista delle sue frasi:

$$D_{fr1} = \{extr(p)\ \forall p \in D_{es}\} \tag{6}$$

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:

$$D_{fr} = \bigcup_{f \in D_{fr1}} f \tag{7}$$

Filtraggio paragrafi Lo scopo di questa fase è quello di rimuovere dal dataset tutti 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 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:

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

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

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

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 3.3, mentre le esatte funzioni per la creazione del dataset sono elencate nella sezione 3.4.
Viene chiamata F_p la lista $[p_1, ..., p_n]$ 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 $[p_1(text_f), ..., p_n(text_f)]$ la cui rappresentazione è schematizzata in Tabella 4.

Codice funzione perturbazione	**Frase perturbata**
$nome\_func_1$	$p_1(f)$
$nome\_func_2$	$p_2(f)$
...	...
$nome\_func_n$	$p_n(f)$

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	Frasi perturbate ($[p_1(text_f), ..., p_n(text_f)]$)

Tabella 5: Struttura di una frase

La fase di perturbazione può quindi essere formalizzata come segue:

$$D_{pe} = \{applyPert(f)\ \forall f \in D_{fp}\} \tag{10}$$

Riduzione Lo scopo della fase di riduzione è quello di creare una versione ridotta del dataset, ovvero di ridurne 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 3.4), il dataset D_{re} è un sottoinsieme di n elementi estratti casualmente da D_{pe}:

$$D_{re} \subset D_{pe} \wedge |D_{re}| = n \tag{11}$$

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

3.3 Perturbazione

Data una stringa s, lo scopo del processo di perturbazione è quello produrre una stringa $s\prime$ 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.

3.3.1 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 $list_t = [t_1, ..., t_n]$, 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:

   $$group : [t_1, ..., t_n] \rightarrow [g_1, ..., g_q] \tag{12}$$

   dove ogni g_i è un gruppo di token $[t_1, ..., t_k]$ 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}

   $$f_{pert} : [t_1, ..., t_k] \rightarrow [t'_1, ..., t'_j] \tag{13}$$

   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 $g \in [g_1, ..., g_q]$ viene perturbato

o meno secondo la seguente funzione:

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

È quindi possibile definire la fase di perturbazione come:

$$pert : [g_1, ..., g_q] \rightarrow [g\prime_1, ..., g\prime_q] \tag{15}$$

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

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

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

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:
$$flat : [g\prime 1, ..., g\prime_q] \rightarrow [t_1, ..., t_m] \tag{17}$$
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 3.3.3 e nella sottosezione 3.3.4.
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 3.3.2.
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:

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

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

Esempi

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

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

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:

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

Assumendo che:

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

è possibile definire $f_{punctsplit}$ come:

$$f_{punctsplit}(t, punct) = \begin{cases} [c1, ..., c_x] \frown punct \frown f_{punctsplit}([c_{x+1}, ..., c_n], punct) & \text{se } x < |t| \\ t & \text{se } x \geqslant |t| \end{cases} \tag{22}$$

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

Esempi

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

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

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:

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

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:

$$f_{pert\_insertion}([\text{"Nel"}], \text{"."}) = [\text{"Nel"}, \text{"."}] \tag{26}$$

$$f_{pert\_insertion}([\text{"cammin"}], \text{"."}) = [\text{"cammin"}, \text{"."}] \tag{27}$$

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

Modulo di hyphen merging

Il modulo di hypen 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:

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

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

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

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:

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

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

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

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 $A_{cammin} =$ [*'cmmin', 'camniin', 'cdmmn', 'camin', 'cammiii'*]. Si sceglie quindi in modo casuale un token dall'insieme A_{cammin}:

$$f_{alternative}(\text{"}cammin\text{"}) = a_{cammin\_2} = \text{"}camniin\text{"} \tag{32}$$

3.3.2 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 3.3, 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 $[t_1, ..., t_n]$ di token:

$$tok : S \to [t_1, ..., t_n] \tag{33}$$

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

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

Modulo di detokenizzazione

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

$$detok : [t_1, ..., t_n] \to S \tag{35}$$

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

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

3.3.3 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 $[t_1, ..., t_n]$ con $n \in \mathbb{N} \wedge n \geqslant 0$), è possibile concatenare più moduli insieme tramite composizione:

$$m_k \circ m_{k-1} \circ m_{k-2} \circ ... \circ m_1 \tag{37}$$

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:

$$Pipeline : detok \circ m_k \circ m_{k-1} \circ m_{k-2} \circ ... \circ m_1 \circ tok \tag{38}$$

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

$$Pipeline(s) = s\prime \tag{39}$$

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.

$$p_{ex} = m_{de} \circ m_{pi} \circ m_{ss} \circ m_{cr} \circ m_{to} \tag{40}$$

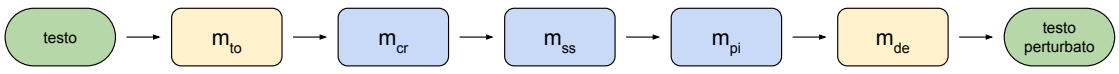


Figura 3: Schema della pipeline p_{ex}

Data una stringa di esempio s_{ex} = *"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}: essendoci nella perturbazione una componente aleatoria, ogni esecuzione di una pipeline su uno stesso input può dare risultati diversi.

3.3.4 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 $Ppl = [p_1, ..., p_n]$ dove ogni p_i è una pipeline,
- Una lista di pesi $W = [w_1, ..., w_n]$ 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:

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

Esempio Si definiscono le tre pipeline $p_{ex1},\ p_{ex1},\ p_{ex1}$, 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.

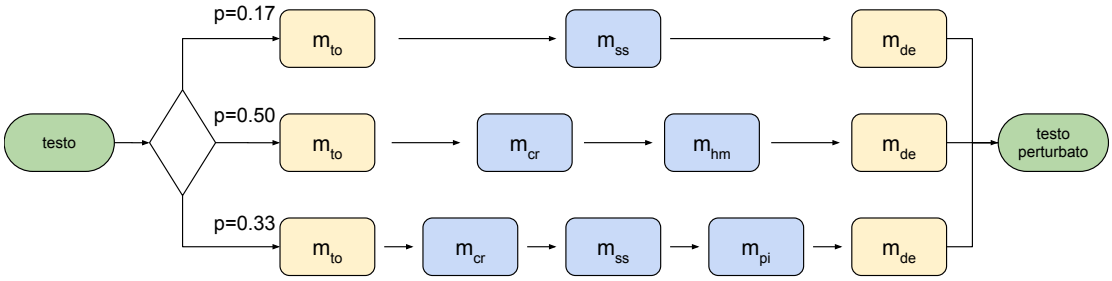


Figura 4: Schema della superpipeline descritta nell'esempio

3.4 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.

3.4.1 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 4.1), 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 4.1), 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 tutto quindi 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 della quali ha li 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.

Codice	$\boldsymbol{p_1}$	$\boldsymbol{p_2}$	$\boldsymbol{p_3}$
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:

Posizione	Tipo Modulo	punct
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.

Codice	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
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.

Codice	Definizione
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

3.4.2 Dataset

In questa sottosezione sono descritte le istanze del dataset create e i loro parametri. Sono ricordati i 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 versione 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 testi, sono approfonditi nel Capitolo 5.

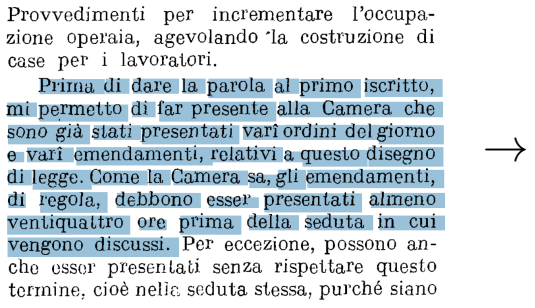
Capitolo 4

Metodologia di correzione

In questo capitolo è descritta la metodologia di correzione.
Nella sezione 4.1 vengono descritti gli obiettivi del processo di correzione e le criticità che lo contraddistinguono. Nella sottosezione 4.2.1 è presente una panoramica generale del processo di correzione. Sono inoltre descritte le fasi e le componenti del sistema. Nella sottosezione 4.2.2 è descritto il funzionamento del BERT Masked Language Model. Nelle sottosezioni 4.2.3 e 4.2.4 è descritto il funzionamento dei moduli che stanno alla base del sistema di correzione.

4.1 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 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 varî 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 var.1 emendamenti, relativi a questo disegno di legge. Come la Camera sa, gli emendamenti, di regola, debbono esser presentati alrneno 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'iinpresa	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	I'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.

4.2 Architettura del sistema

4.2.1 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 errori all'interno di una frase. Un modulo può essere formalmente definito come segue:

$$M : F \mapsto F\prime \tag{42}$$

dove F e $F\prime$ 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 4.2.3.
- **Modulo di correzione Split** (M_{SP}): individua e corregge tutti gli errori di tipo SP. È discusso in dettaglio nella sottosezione 4.2.4.

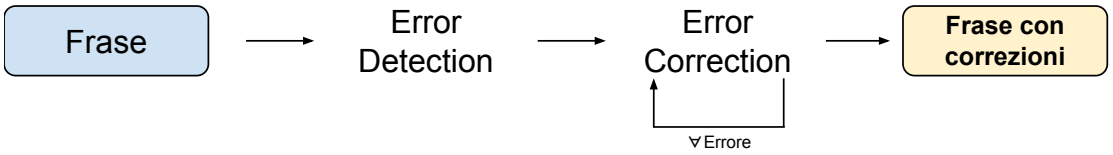


Figura 6: Schema del funzionamento di un modulo

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 dalla 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:

$$P : f_1 \circ f_2 \circ ... \circ f_n \tag{43}$$

dove ogni f_i è un modulo.

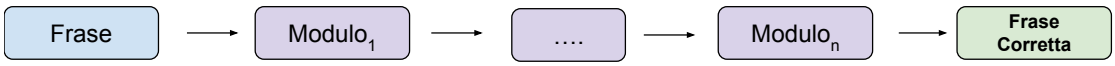


Figura 7: Schema riassuntivo della pipeline di correzione

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.

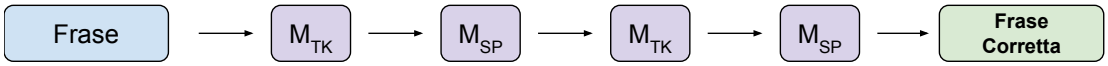


Figura 8: Esempio di una possibile pipeline con più occorrenze di uno stesso modulo

4.2.2 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 $S = [w_1, ..., w_n]$ 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 $S\prime = [w_1, ..., [\text{MASK}], ..., w_n]$.
Dando in input $S\prime$ al modello BERT, esso associa ad ogni parola del proprio lessico la probabilità di corrispondere la 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":

$$B : S\prime \rightarrow [c_1, ..., c_n] \tag{44}$$

dove ogni $c_i \in [c_1, ..., c_n]$ 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 mascherala, 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 inferite unicamente dal contesto derivato dal resto della frase.

4.2.3 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 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 seguente 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 di modulo di correzione Split (sottosezione 4.2.4).

- 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.

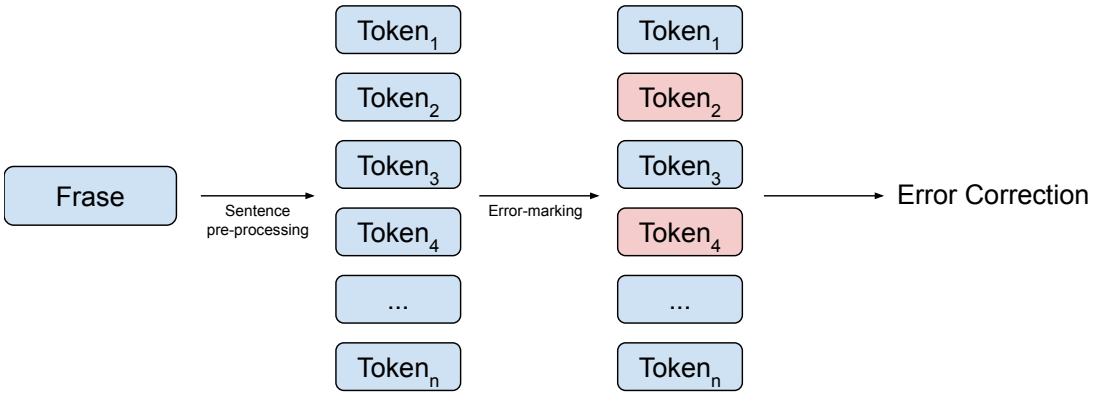


Figura 9: Schema del processo di error detection

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.

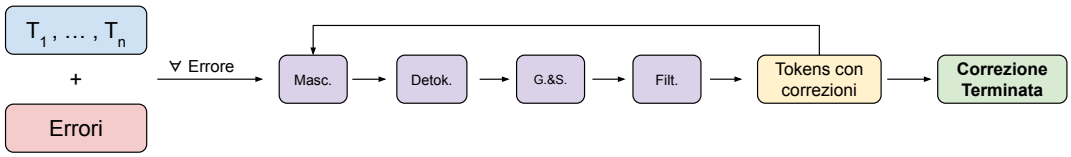


Figura 10: Schema della fase di error correction

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

$$M : S \to [t_1, ..., [\text{MASK}], ..., t_n] = S\prime \tag{45}$$

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

Detokenization Come spiegato nella sottosezione 4.2.2, 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:

$$D : [t_1, ..., [\text{MASK}], ..., t_n] \to F_{mask} \tag{46}$$

dove F_{mask} è una frase contente 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 4.2.2). 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%. TODO: inserire riferimento ai risultati del capitolo analisi errore.
Lo step di generazione dei candidati è formalizzato come segue:

$$G : F_{mask} \rightarrow [c_1, ..., c_n] = C \tag{47}$$

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:

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

In caso uno o più candidati abbiano la stessa distanza dal token mascherato, viene scelto quello con la maggior probabilità prodotta da BERT. TODO: giustificazione dell'euristica attraverso risultato presenti nel capitolo di analisi errore

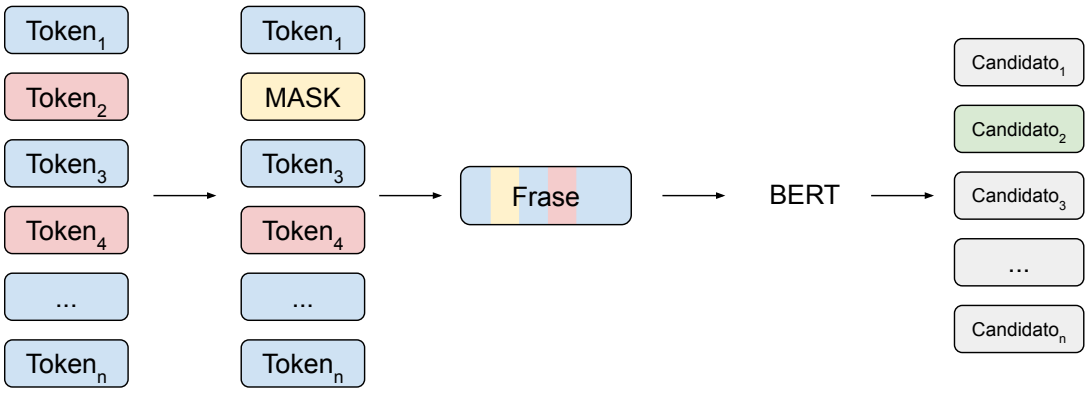


Figura 11: Schema del processo di generazione e scelta dei candidati

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:

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

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.

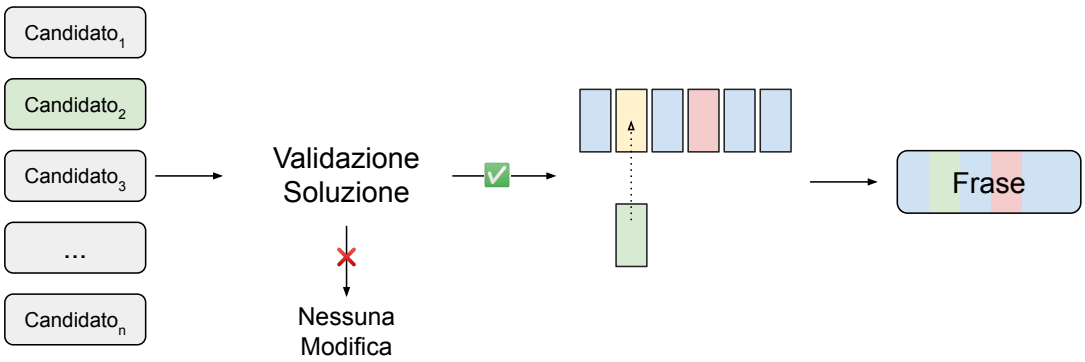


Figura 12: Schema del processo di scelta dei candidati

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 della prime correzioni sono sbagliate, il rischio è quello di introdurre rumore anche nelle correzioni successive.

4.2.4 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 un 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'(?:\s|^)(\w(?:\W?\s\w{1,2}){3,})(?:\W|\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 e 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 ritenga 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	*"d i l e t t i membri, della ..."*	*d i l e t t i*	Sì
2	*"guidato d ui l l a fede"*	*d ui l l a*	Sì
3	*"... altro tipo di u n i o, n e."*	*u n i o, n e*	Sì
4	*"in me e io in te, siano anch..."*	*in me e io in te*	No
5	*"...dopo qu al ch e esitazione..."*	*qu al ch e*	Sì
6	*"...dopo q u al che esitazione..."*	*q u al*	Sì
7	*"...dopo qu al che esitazione..."*	/	/

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:

$$ED_{split} : F \rightarrow L_{seq} \tag{49}$$

dove ogni $l_{seq} \in L_{seq}$ è un insieme di tuple $[(s_1, b_1, e_1), ..., (s_n, b_n, e_n)]$ 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 delle frase 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:

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

dove F_{corr} è la frase con le correzione 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 $s\prime_i$.
2. **Vocabulary correction**: se una sequenza $s\prime_i$ è presente nel vocabolario, allora è $s\prime_i$ 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 $s \in S$, una valore di inizio $b \in I$ e un valore di fine $e \in E$, la funzione è definita come:

$$Sos : (F, S, I, E) \rightarrow F\prime \tag{51}$$

dove $F\prime$ 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:

$$Sos : (F, T) \rightarrow F\prime \tag{52}$$

Ad esempio, data la frase f_{es}:

Onorando una trod1z1one plurisecolare

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

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

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:

$$Comp : T \rightarrow T\prime \tag{54}$$

dove in cui $T\prime$ è l'insieme di tutte le tuple $(s\prime, b, e)$ in cui $s\prime$ 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 $(s\prime, b, e)$ detta t_{corr} che rispetta la condizione enunciata precedentemente, la frase con la correzione applicata F_{corr} si ottiene come segue:

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

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:

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

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 fase originale. Data una tupla $(s\prime, b, e)$ prodotta dallo stadio di compressione e una frase F, si ottiene la frase mascherata F_{mask} come segue:

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

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 4.2.3, 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 $s\prime$. 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 $s\prime$, si ottengono le due stringhe sub_1 e sub_2 come segue:

$$sub_1, sub_2 = c_{corr}, s\prime - c_{corr} \tag{58}$$

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

$$sub_1, sub_2 = s\prime - c_{corr}, c_{corr}, \tag{59}$$

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

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

La correzione con sottrazione corregge i casi in cui la sequenza contenente errore è formata da due parole. Può accadere che la stringa $s\prime - c_{corr}$ 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:

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

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 $s\prime$, pertanto si ottengono $sub_1, sub_2 = e, ogni$. Si applica quindi la funzione di sostituzione:

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

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 $s\prime$. Questo step utilizza un'euristica per validare la correzione prodotta dal BERT MLM. La soluzione prodotta da BERT è valutata ammissibile e solo se $d_{lev}(s\prime, c_{corr}) \leqslant 4$.
In caso la soluzione sia ammissibile, la correzione è applicata tramite la funzione di sostituzione:

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

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

Capitolo 5

Implementazione, Test e Risultati

Capitolo 6

Analisi dell'errore

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 differance 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 Hämäläinen and Simon Hengchen. From the paft to the fiiture: 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 Hämäläinen, 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.