ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

SUMMARIZATION ASTRATTIVA DI LUNGHI ARTICOLI SCIENTIFICI MEDIANTE ESTRAZIONE DI FRAMMENTI RILEVANTI

Elaborato in
Programmazione Di Applicazioni Data Intensive

Relatore
Prof. Gianluca Moro

Co-relatori
Dott. Paolo Italiani
Dott. Luca Ragazzi

Presentata da
Filippo Di Pietro

Terza Sessione di Laurea
Anno Accademico 2022 – 2023

PAROLE CHIAVE

Natural Language Processing

Text Summarization

Transformers

Machine Learning

Deep Neural Networks

A chiunque mi sia stato vicino,
e mi abbia aiutato a raggiungere questo traguardo.

Abstract

L'obiettivo di questo lavoro, nel contesto della text summarization, evidenziare le limitazioni degli attuali language model nel concentrarsi sulle parti rilevanti di un documento. Coinvolgendo la specializzazione di un modello nella generazione di sequenze di testo o frasi comuni tra l'articolo originale e il suo riassunto. Attraverso esperimenti condotti utilizzando il dataset di arXiv [Cohan et al., 2018], e SciLay. Il lavoro metterà in luce che, pur esistendo opportunità di apportare miglioramenti significativi in alcuni contesti, tali miglioramenti rimangono fuori portata a causa di alcune restrizioni nei modelli attuali. La struttura della tesi prevede l'analisi delle diverse tecniche di estrazione del testo comune, la definizione dei vari algoritmi impiegati, il loro potenziale di miglioramento e la valutazione delle performance di ciascun modello mediante il fine tuning in ogni fase del processo.

Indice

Elenco delle figure

Capitolo 1

Introduzione

L'Intelligenza Artificiale, o AI, è una disciplina che si concentra sulla creazione di sistemi, programmi informatici o modelli in grado di compiere compiti che richiedono intelligenza umana, tra cui: il riconoscimento d'immagini, il riconoscimento del linguaggio naturale, la pianificazione, la risoluzione di problemi complessi e molto altro ancora. L'obiettivo fondamentale dell'AI è emulare la capacità umana di apprendere, ragionare e prendere decisioni in modo autonomo.

L'AI risale almeno agli anni '50 del XX secolo. Inizialmente, gli sforzi si concentravano su programmi d'intelligenza simbolica che utilizzavano regole e simboli per risolvere problemi. Tuttavia, negli anni '80, l'AI ha affrontato una fase di stasi nota come "inverno dell'IA", in cui le aspettative superavano la realtà delle capacità dei computer dell'epoca.

Negli ultimi due decenni, l'AI ha vissuto una rinascita spettacolare grazie a importanti avanzamenti tecnologici. Il deep learning, una branca dell'AI che utilizza reti neurali artificiali profonde, ha rivoluzionato il campo. L'uso di algoritmi di apprendimento automatico in combinazione con l'abbondante potenza di calcolo ha permesso a computer e sistemi di apprendere da enormi quantità di dati e migliorare le prestazioni in una vasta gamma di compiti.

L'AI ha trovato applicazioni innumerevoli in molte industrie e settori. Ad esempio, nell'ambito della medicina, l'AI è utilizzata per la diagnosi di malattie, l'analisi d'immagini mediche e la scoperta di nuovi farmaci. Nel settore automobilistico, sta contribuendo allo sviluppo di veicoli autonomi. Nel campo finanziario, l'AI è utilizzata per il trading automatizzato e la gestione del rischio, e questi sono solo alcuni esempi.

Nonostante i progressi straordinari, l'AI affronta ancora alcune sfide significative, una di queste è la trasparenza e l'interpretabilità dei modelli di AI, specialmente quando si tratta di decisioni critiche per la vita umana. Ci sono anche discussioni in corso sulle implicazioni etiche dell'AI, come la privacy dei

dati e l'equità nell'accesso alle tecnologie basate sull'AI.

L'AI è in continua evoluzione, e il suo futuro è entusiasmante, con gli scienziati e gli ingegneri che stanno lavorando su nuove sfide. In un campo specifico, come quello dell'elaborazione del linguaggio naturale (NLP), una delle sfide più pressanti è la Text Summarization. Essa richiede l'abilità di estrarre e condensare in modo accurato e coerente le informazioni più rilevanti da testi estesi e complessi. Nell'era digitale in cui il flusso d'informazioni è in costante aumento, la capacità di sintetizzare testo in modo efficiente è diventata cruciale per un'ampia gamma di applicazioni, dall'accesso rapido alle notizie all'estrazione di conoscenze da documenti tecnici.

Una delle sfide principali in questo campo è l'ambiguità intrinseca del linguaggio. Il linguaggio umano è incredibilmente vario e ricco di sfumature, dove una vasta gamma di fenomeni linguistici e stili di scrittura può essere utilizzata per esprimere lo stesso concetto, tale ambiguità linguistica può creare problemi significativi per i sistemi automatizzati di Text Summarization, ad esempio: diverse parole possono avere significati multipli, e la stessa informazione può essere espressa in modi diversi da autori diversi.

Nonostante i notevoli progressi raggiunti grazie alle tecnologie di deep learning nel campo dell'NLP, ci sono sfide persistenti che richiedono ancora soluzioni innovative. Anche i modelli di linguaggio basati su transformer con centinaia di miliardi di parametri, che rappresentano una pietra miliare nell'evoluzione dell'NLP, affrontano difficoltà nel distinguere le relazioni semantiche chiare dalle strutture superficiali del linguaggio. Infatti, spesso porta a risultati indesiderati come "allucinazioni", cioè la generazione di contenuti inesistenti, e "fragilità", ovvero la vulnerabilità a manipolazioni esterne.

In particolare, nel campo della Text Summarization, è fondamentale lavorare a un livello semantico più profondo, ciò significa andare oltre la mera manipolazione di parole e frasi e cercare di catturare il significato sottostante. Tale necessità è ancora più evidente nella letteratura biomedica, dove le informazioni devono essere precise e la tolleranza per errori è veramente molto bassa. In questo contesto, l'elaborazione semantica accurata è essenziale per comprendere i testi in modo corretto.

La comunità di ricerca sta rispondendo a queste sfide identificando la necessità d'integrare conoscenze multidimensionali. Ciò potrebbe avvenire, ad esempio, attraverso l'uso di grafi di conoscenza esterna o rappresentazioni strutturate ottenute tramite analisi semantica. Inoltre, si sta esplorando l'estrazione di eventi come una promettente direzione per migliorare la qualità delle sintesi del testo. La combinazione di queste diverse tecniche è fondamentale per affrontare la complessità dell'ambiguità linguistica e generare sintesi di alta qualità.

In questa tesi incentrata sulla Text Summarization, si pone l'obiettivo di aumentare le performance dei modelli esistenti, usando una strategia per

processare solo il testo più rilevante in documenti lunghi, specializzando un modello nella generazione di sequenze di testo o frasi comuni tra articolo e riassunto gold, per poi specializzare un altro modello nella generazione finale del riassunto, cercando quindi di migliorare la sintesi astrattiva dei documenti. Attraverso esperimenti sul dataset di arXiv [Cohan et al., 2018], si dimostrerà che in alcuni casi si possono ottenere dei miglioramenti. La tesi sarà strutturata in modo da esaminare le diverse tecniche di estrazione del testo comune, definire i vari algoritmi usati, come si possono migliorare, e la performance di un modello, eseguendo il fine tuning per ogni step della pipeline. In questo modo, ci immergeremo nel complesso mondo della Text Summarization, esplorando le sfide e le soluzioni in continua evoluzione in questo campo cruciale dell'NLP.

1. **Capitolo 1** - Introduzione;
2. **Capitolo 2** - Analisi preliminare sui modelli che hanno raggiunto lo stato dell'arte nella Text Summarization;
3. **Capitolo 3** - Estrazione delle frasi o frammenti comuni tra articolo e riassunto, e valutazione di essi;
4. **Capitolo 4** - Conclusione.

Capitolo 2

Text Summarization - Stato dell'Arte

Lo stato dell'arte nel campo della Text Summarization ha fatto progressi significativi negli ultimi anni dovuti all'uso di modelli di linguaggio basati su reti neurali profonde, soprattutto i modelli di tipo transformer, come BERT (Bidirectional Encoder Representations from Transformers), hanno cambiato il modo in cui affrontiamo il problema di text summarization. Questi modelli riescono a catturare le relazioni complesse tra le parole e a generare riassunti di testo che sono più coerenti e informativi rispetto agli approcci tradizionali.

Il task di text summarization è suddiviso in due categorie principali: l'estrazione di frasi chiave (*extractive*) e la generazione di riassunti (*abstractive*). La categoria di estrazione, si concentra sull'identificare le frasi più importanti nel testo originale e selezionarle per il riassunto. La categoria di generazione, invece, si concentra sul generare il riassunto con parole che non si trovano nel testo originario, richiede quindi delle operazioni di riscrittura. Esistono anche approcci ibridi, combinando quindi estrazione e astrazione. Alcuni modelli di text summarization sono stati specializzati su specifici domini, come le notizie o la medicina, per ottenere risultati ancora più precisi e coerenti in determinati contesti. Un esempio pratico di questo approccio è EASE di Facebook, che usando l' Information Bottleneck (IB) sono riusciti a superare anche se di poco BERTSUM sul dataset CNN/DailyMail e di 4 punti sul dataset Xsum, ma non sono riusciti a superare lo stato dell'arte mantenuto da BART-large, come è possibile vedere in Tabella 2.3. Di seguito i principali modelli e tecniche usate nella text summarization:

- LEAD è una tecnica di estrazione di frasi basata su una euristica semplice: vengono selezionate le prime tre frasi (o un altro numero fisso di frasi) del testo di origine come riassunto. Questa tecnica viene utilizzata spesso come baseline per la valutazione di modelli più avanzati.

- REFRESH [Narayan et al., 2018] è un modello estrattivo basato sull'ottimizzazione globale, o su una finestra di frasi più grande, della metrica ROUGE mediante apprendimento per rinforzo, invece di ottimizzare localmente solo una singola frase alla volta. Questo approccio aiuta a garantire che le frasi selezionate si integrino bene tra loro nel riassunto e che il riassunto nel suo complesso sia coerente e informativo.

- NEUSUM [Zhou et al., 2018] è un modello estrattivo sviluppato da. Questo sistema è noto per essere uno degli stati dell'arte nell'ambito della text summarization estrattiva, al momento della sua pubblicazione, che valutano e selezionano le frasi per la creazione di riassunti. Assegna un punteggio a ciascuna frase nel documento di origine. Questo punteggio è basato su vari criteri, tra cui la rilevanza, l'importanza e altri fattori che determinano quanto una frase sia adatta a essere inclusa nel riassunto. NEUSUM adotta un approccio congiunto in cui considera il punteggio di ogni frase in relazione alle altre frasi nel testo di origine.

- PGN [See et al., 2017] è un modello per sintesi astrattiva che si basa su un'architettura encoder-decoder, la quale è comune nei modelli di generazione di testo. L'encoder processa il testo di origine e cattura le informazioni chiave, mentre il decoder genera il riassunto in base alle informazioni acquisite dall'encoder. Inoltre, utilizza meccanismi di attention, consentendogli di concentrarsi su parti specifiche del testo di origine. Questo modello è noto per la sua capacità di generare riassunti che possono contenere parole e frasi non presenti nel testo di origine, rendendolo un sistema di generazione di riassunti astrattivi.

- DCA [Celikyilmaz et al., 2018] è un modello per sintesi astrattiva. Utilizza un approccio con agenti di comunicazione profonda per rappresentare il testo di origine. Gli agenti sono moduli che interagiscono tra loro per catturare informazioni rilevanti dal documento sorgente in modo cooperativo. Questa rappresentazione distribuita aiuta il modello a comprendere meglio il contesto del documento. Inoltre, implementa un meccanismo di attenzione gerarchica che consente al modello di concentrarsi su diverse parti del documento in modo sequenziale e strutturato. Ciò aiuta a gestire documenti lunghi e a catturare relazioni complesse tra le frasi.

- T5 (Text-to-Text Transfer Transformer) [Raffel et al., 2023] è un modello di linguaggio basato su transformers. È noto per la sua architettura innovativa che tratta i compiti di NLP come problemi di conversione da testo a testo. Questo approccio lo rende altamente versatile, in quanto può essere utilizzato per una vasta gamma di compiti di NLP, compresi

il riassunto, la traduzione, la generazione di testo, la risposta a domande e molto altro.

- UniLM (Unified Language Model) [Dong et al., 2019] è un modello di linguaggio progettato per essere un modello unificato che può affrontare una serie di compiti di NLP, inclusi il riassunto di documenti, la traduzione automatica, il completamento automatico di testi e la risposta a domande. Questa unificazione consente al modello di condividere informazioni tra compiti diversi e di apprendere rappresentazioni linguistiche più generali. Il modello utilizza un'architettura bidirezionale simile a BERT, consentendogli di catturare informazioni da entrambi i lati di una sequenza di testo. Questo lo rende efficace nella comprensione del contesto. UniLM ha ottenuto risultati di punta su una varietà di compiti di NLP (tra cui la text summarization) ed è noto per la sua flessibilità e capacità di eseguire sia compiti di generazione di testo che compiti di comprensione del linguaggio.

- MASS (Masked Sequence to Sequence pre-training) [Song et al., 2019] si basa su un'architettura di tipo encoder-decoder. L'encoder riceve una sequenza di token con una parte casuale della sequenza mascherata (cioè sostituita con token di maschera) come input, mentre il decoder cerca di prevedere la parte mascherata. Il compito principale durante il pre-training è la previsione dei token mascherati. In altre parole, l'encoder crea una rappresentazione del contesto per la parte mascherata e il decoder cerca di prevedere esattamente quella parte. Questo compito di previsione è analogo al riempimento di spazi vuoti in un testo. Il pre-training di MASS è effettuato su un corpus di testi molto vasto senza l'uso di supervisione specifica per compiti come la traduzione o la sintesi. Ha ottenuto risultati di punta nella traduzione non supervisionata da inglese a francese, superando persino i modelli supervisionati basati su attention.

- SEAL [Zhao et al., 2020] è un modello di linguaggio specializzato nel riassunto di testi lunghi che combina metodi estrattivi e astrattivi, codificando e tenendo conto della segmentazione del testo. SEAL divide il processo di decodifica in segmenti non sovrapposti, ciascuno con una dimensione definita. Diverse porzioni di testo vengono selezionate per il decodificatore da utilizzare in ciascun segmento di codifica. All'inizio di ciascun segmento di decodifica, l'encoder codifica i token di ciascun segmento precedente in rappresentazioni. Questa codifica tiene conto dei segmenti di decodifica precedenti per rendere il modello consapevole di ciò che è stato decodificato. Inoltre utilizza un meccanismo di attenzione tra

encoder e decoder che cambia dinamicamente tra i segmenti di decodifica. Questa selezione dinamica delle porzioni di testo consente un utilizzo più efficiente della memoria, migliori etichette proxy (calcolate sulla base delle somiglianze tra ciascun segmento del riassunto di riferimento e le porzioni di testo di input) e una maggiore interpretabilità.

- BERTSUM [Liu, 2019] è un modello per sintesi estrattiva basato sull'utilizzo del modello preaddestrato BERT, sfruttando le rappresentazioni contestuali apprese da BERT per generare riassunti coerenti e significativi di documenti o testi lunghi. Inizialmente, il documento o il testo da riassumere viene suddiviso in frasi o blocchi di testo più piccoli, successivamente viene quindi codificato utilizzando BERT, questo processo di codifica cattura le rappresentazioni contestuali delle parole e delle frasi nel testo. Per identificare le frasi più rilevanti da includere nel riassunto, viene calcolato un punteggio di rilevanza per ciascuna frase rispetto all'intero testo. Il punteggio può essere calcolato utilizzando diverse metriche, ma BERTSUM utilizza la similarità coseno tra le rappresentazioni delle frasi. Le frasi con i punteggi di rilevanza più alti vengono selezionate per formare il riassunto. La quantità di frasi selezionate può variare a seconda delle preferenze o dei requisiti specifici, ma solitamente è un sottoinsieme delle frasi originali. Una volta selezionate le frasi più rilevanti, queste vengono concatenate per formare il riassunto finale. La concatenazione delle frasi può richiedere una post-elaborazione per garantire la coerenza e la leggibilità del riassunto. Infine, il riassunto può essere sottoposto a ulteriori passaggi di valutazione e raffinamento per assicurarsi che rispetti gli obiettivi del riassunto, come la concisione, l'accuratezza e la chiarezza. BERTSUM è stato utilizzato in vari contesti, tra cui il riassunto automatico di notizie, documenti giuridici e testi accademici.

  - BERTSUM + Classifier [Liu, 2019] è una tecnica che combina BERTSUM con un classificatore aggiuntivo per condurre ulteriori analisi o classificazioni basate sui riassunti generati. Inizialmente, si utilizza l'approccio BERTSUM per generare un riassunto del testo di input, BERTSUM seleziona le frasi più rilevanti e crea un riassunto coerente e significativo del testo. Il riassunto generato viene quindi passato attraverso un classificatore noto come "Simple Classifier". Questo classificatore consiste in un singolo strato lineare che opera sulle rappresentazioni del riassunto e utilizza una funzione di attivazione sigmoidale per ottenere un punteggio predetto per una determinata classe o etichetta, valore che è compreso tra 0 e 1, che può essere interpretato come la probabilità di appartenenza a tale classe o etichetta.

  - BERTSUM + Transformer [Liu, 2019] è una combinazione che sfrutta la capacità di BERT di comprendere il contesto del testo e l'efficienza del Transformer nell'elaborare sequenze. La seguente combinazione mantiene l'encoding tramite BERT, con la differenza che invece di utilizzare un semplice classificatore sigmoidale, l'approccio seguente applica ulteriori strati di Transformer solo alle rappresentazioni delle frasi, estraendo così delle caratteristiche a livello documento concentrandosi sui compiti di riassunto, utilizzando le uscite di BERT come input. Per tener conto delle posizioni delle frasi nel documento, vengono aggiunte degli embedding di posizione (PosEmb) alle rappresentazioni delle frasi. Questi indicano la posizione di ciascuna frase nel documento. Le rappresentazioni delle frasi vengono ulteriormente aggiornate attraverso un livello feed-forward (FFN). Infine avviene la classificazione finale tramite un classificatore sigmoidale che viene applicato per produrre una previsione o una classificazione finale basata sul riassunto generato. Nel corso degli esperimenti, sono stati implementati modelli Transformer con diversi numeri di strati (L). Dai risultati ottenuti, sembra che un modello con 2 strati Transformer (L = 2) abbia dato le migliori performance nei compiti di riassunto.
  - BERTSUM + LSTM [Liu, 2019] il quale facendo uso della **L**ong **S**hort-**T**erm **M**emory estrae le frasi in un ordine coerente.

- BART [Lewis et al., 2019] (Bidirectional and Auto-Regressive Transformers) è un modello di linguaggio basato su una variante del Transformer che è stato progettato per l'elaborazione del linguaggio naturale e ha dimostrato essere molto versatile e potente in una varietà di compiti di NLP. A differenza di alcuni modelli di linguaggio che sono o bidirezionali (come BERT) o auto-regressivi (come GPT), BART combina entrambi gli approcci potendo così gestire input sia in modalità bidirezionale, dove ha accesso a tutto il contesto, sia in modalità auto-regressiva, dove genera un token alla volta in modo sequenziale. Può essere utilizzato per generare riassunti, l'analisi del sentimento, tradurre lingue, o completare frasi.

- PEGASUS [Zhang et al., 2020] (Pre-training with Extracted Gap-sentences for Abstractive Summarization) è noto per essere un sistema di riassunto astrattivo estremamente potente e versatile, basato sulla popolare architettura dei Transformer con 223 Milioni di parametri per il modello base. Il modello utilizza una tecnica di tokenization specifica chiamata "sentencepiece", che suddivide il testo in token basati su frasi piuttosto che su singole parole. Questo aiuta il modello a gestire meglio la struttura

delle frasi nei riassunti. Pegasus fa uso di frasi estratte dal testo di origine, chiamate "gap-sentences", le quali durante la fase di pre-training vengono mascherate e il modello è addestrato per prevederle. Questo approccio contribuisce a migliorare la capacità di Pegasus di riassumere il testo di origine.

  - PEGASUS Large, a differenza del modello base, possiede 568 milioni di parametri, 16 layer (contro i 12 del base) di encoder e decoder, 16 self-attention heads (invece che 14 rispetto al base), 1024 dimensioni degli strati nascosti (invece che 768), 4096 dimensioni del livello feed-forward (invece che 3072), infine un batch size di 8192 (invece che 256).
  - Esistono anche delle versione di PEGASUS che sono state pre-addestrate solo sui dataset C4 e HugeNews, ottenendo così delle performance ancora più superiori alla versione base di PEGASUS.

- EASE [Li et al., 2021] (Extractive-Abstractive Summarization with Explanation) combina tecniche di riassunto estrattivo e astrattivo per affrontare le limitazioni dei sistemi esistenti. Il modello è strutturato in due parti principali: estrattore, astrattore. Estrattore è responsabile dell'estrazione delle informazioni salienti dal documento di origine, basato su un transformer preaddestrato, simile a BERT. Tuttavia, è dotato di un ulteriore strato lineare che calcola delle probabilità per ciascun token nel documento originale, la quale rappresenta la possibilità che venga selezionato (ossia, non mascherato) durante il processo di estrazione. L'obiettivo dell'estrattore è identificare quali parti del testo sono importanti. L'astrattore invece, è la parte del modello è responsabile della generazione del riassunto finale, basato su un modello di linguaggio seq-to-seq preaddestrato. Simile a BART, prende in input le evidenze estratte dall'estrattore e genera il riassunto del testo sulla base di queste evidenze.

- DANCER [Gidiotis and Tsoumakas, 2020] affronta il problema dei documenti lunghi suddividendoli in sezioni distinte e riassumendo ciascuna sezione separatamente, assumendo che i documenti lunghi siano suddivisi in sezioni discrete, e si sfrutta questa struttura del discorso lavorando su ciascuna sezione in modo separato. Durante l'addestramento del modello, ogni sezione del documento viene trattata come un esempio diverso, accoppiandola con un riassunto distinto come obiettivo. Per accoppiare ciascuna parte del riassunto con una sezione del documento, si utilizzano le metriche ROUGE, che misurano la somiglianza tra le sequenze di

parole. L'architettura del modello Pointer-Generator è una rete neurale sequenza-a-sequenza che combina un encoder, un decoder e meccanismi di attenzione e copia. L'encoder è responsabile di codificare l'input, in una rappresentazione nascosta. In questo caso, l'encoder utilizza una serie di unità LSTM bidirezionali per elaborare l'input, queste unità LSTM possono considerare il contesto sia a sinistra che a destra di ogni parola nell'input, producendo una sequenza di stati nascosti dell'encoder. Il decoder è una rete LSTM unidirezionale che genera l'output, inizialmente, lo stato nascosto del decoder viene inizializzato utilizzando un vettore di contesto predefinito. Successivamente, il decoder genera le parole una alla volta in modo auto regressivo. Ad ogni passo temporale t, il decoder riceve in input lo stato nascosto dell'encoder h, la parola generata al passo precedente e produce uno stato nascosto. Durante l'addestramento, ciascuna sezione del documento viene utilizzata come testo di input e la parte corrispondente del riassunto è il riassunto obiettivo. L'addestramento viene eseguito utilizzando il teacher forcing, un metodo di addestramento che minimizza la negativa log-likelihood delle sequenze obiettivo dati le sequenze di input. Questo metodo ha diversi vantaggi, tra cui una maggiore efficienza nel trattare con documenti lunghi, una distribuzione più equa delle sequenze di input e un accoppiamento più accurato tra le sequenze di input e di output. Inoltre, riduce la complessità computazionale. Non tutte le sezioni di un documento sono importanti per il riassunto, ad esempio, le sezioni di revisione della letteratura possono essere meno rilevanti. Pertanto, viene effettuata una selezione delle sezioni in base a parole chiave nei titoli delle sezioni. Un limite di questo metodo è che alcune sezioni potrebbero non essere abbinate in modo appropriato. Si suggerisce di esplorare metodi più sofisticati basati sull'apprendimento automatico per l'identificazione dei tipi di sezioni.

- DANCER + RUM comporta alcuni vantaggi. In particolare, le RUM generano gradienti più grandi durante l'addestramento, il che rende il processo di addestramento più stabile e migliora la convergenza del modello. Questo è in contrasto con le unità LSTM, le cui porte di attivazione di solito hanno funzioni di attivazione tangente iperbolica (tanh), e di conseguenza i gradienti diventano molto rapidamente piccoli, anche quando si utilizza il "gradient clipping" (limitazione dei gradienti per evitare problemi di esplosione del gradiente). Nel contesto del riassunto, è stata creata una variante del modello in cui le unità LSTM nel decodificatore sono state sostituite dalle unità RUM. Tuttavia, le unità LSTM sono state mantenute nell'encoder. Questa scelta è stata fatta perché è stato dimostrato che una combinazione

di entrambi i tipi di unità (LSTM e RUM) può essere vantaggiosa per il processo di riassunto.

- DANCER + PEGASUS si basa sulla trasformazione di sequenza-a-sequenza ed è stata preaddestrata su un vasto corpus di dati non supervisionato costituito da articoli web e notizie. Questa variante utilizza DANCER per suddividere un documento di testo lungo in segmenti più piccoli o sezioni significative, il risultato passa in input a PEGASUS, il quale utilizza un modello Transformer di tipo encoder-decoder ed è in grado di generare riassunti a partire da un testo di input. Dopo il preaddestramento, il modello può essere ulteriormente addestrato specificamente per compiti di riassunto. Questo processo di fine-tuning è importante perché adatta il modello al compito specifico di generare riassunti da testi di input. Durante il preaddestramento, il modello utilizza un obiettivo di addestramento auto-supervisionato chiamato "Gap Sentence Generation" (GSG). In questo obiettivo, vengono mascherate intere frasi da un documento e il modello deve generare queste frasi mancanti a partire dal resto del documento. Questo incentiva il modello a comprendere l'intero documento e a generare frasi in modo simile a un riassunto. Dopo il preaddestramento, il modello può essere ulteriormente addestrato specificamente per compiti di riassunto. PEGASUS introduce una strategia per selezionare le frasi importanti da mascherare nel documento durante l'obiettivo GSG. Questo è fatto in modo mirato, anziché selezionare frasi casuali, per migliorare l'addestramento del modello. A differenza di alcuni altri modelli, come il Pointer-Generator model, PEGASUS opera a livello di token subword invece di token word. Questo significa che il testo viene diviso in unità più piccole rispetto alle parole. Questa pratica è comune in molti modelli Transformer ed è possibile grazie all'uso di un vocabolario basato sull'algoritmo Unigram di SentencePiece. Questa scelta consente al modello di apprendere e utilizzare una vasta gamma di parole con un vocabolario limitato. Grazie all'uso di token subword e alla varietà del vocabolario appreso, il modello PEGASUS non richiede l'impiego di meccanismi di copia (copying mechanisms) come quelli utilizzati in alcuni altri modelli di riassunto. Questo significa che il modello è in grado di generare il testo di riassunto senza bisogno di copiare direttamente le parole dal testo di origine.

- LED (Longformer-Encoder-Decoder) [Beltagy et al., 2020] è una versione modificata del modello Transformer, progettata per gestire sequenze di input molto lunghe in modo più efficiente. La principale innovazione

di Longformer è l'uso di un "pattern di attenzione" che specifica quali coppie di posizioni di input dovrebbero essere considerate nell'autorappresentazione (self-attention). A differenza dell'auto-attenzione completa nel modello Transformer originale, questo pattern di attenzione scala linearmente con la lunghezza della sequenza di input, rendendolo efficiente per sequenze più lunghe. Il pattern di attenzione di Longformer utilizza una finestra scorrevole di dimensioni fisse intorno a ciascun token. Questa finestra scorrevole consente a ciascun token di considerare solo una porzione limitata di token circostanti. Inoltre, possono essere impilate diverse "finestre scorrevoli" per aumentare il campo di ricezione, consentendo ai livelli superiori di accedere a tutte le posizioni di input. Per aumentare ulteriormente il campo di ricezione senza aumentare la complessità computazionale, la finestra scorrevole può essere "dilatata". Questo significa che la finestra ha intervalli di gap tra le posizioni considerate. Questa dilatazione è simile a come vengono utilizzati i convoluzionali dilatati nelle reti neurali convoluzionali (CNN). Longformer introduce anche un'attenzione globale su alcune posizioni di input preselezionate. Questa attenzione globale è simmetrica, il che significa che un token con attenzione globale considera tutti gli altri token nella sequenza e viceversa. Questa attenzione globale può essere utilizzata per modellare rappresentazioni specifiche del compito. Per calcolare lo score di attenzione, Longformer utilizza due set di proiezioni lineari separate: uno per l'attenzione locale (finestre scorrevoli) e uno per l'attenzione globale. Queste proiezioni forniscono flessibilità per modellare tipi di attenzione diversi in modo ottimale.

- Top Down Transformer [Pang et al., 2022] è un modello di tipo Transformer progettato per la generazione di sommari da documenti. Il modello utilizza un approccio ibrido che combina l'elaborazione "bottom-up" e "top-down" per creare rappresentazioni dei token in ingresso più efficaci ed efficienti. Nella fase di Bottom-Up , vengono calcolate le rappresentazioni dei token di input utilizzando N1 strati di self-attention locale. Ogni token considera solo i token vicini all'interno di una finestra di dimensione w, rendendo la self-attention locale più efficiente in termini di tempo e memoria rispetto all'auto-attenzione completa, poiché ha una complessità di $O(Nw)$ anziché $O(N^2)$. La fase di Top-Down invece affronta la limitazione delle rappresentazioni locali, aggiornando le rappresentazioni dei token "bottom-up" con informazioni globali, utilizzando una struttura multi-scala con livelli superiori per rappresentare informazioni a livello di documento. Questi livelli superiori utilizzano l'auto-attenzione completa, consentendo loro di catturare il contesto globale del documento. L'aggiornamento delle rappresentazioni durante la fase di "top-down"

avviene aggiornando l'informazione globale tramite un'operazione di auto-attenzione incrociata tra i token bottom-up e i token top-down. Questo permette alle rappresentazioni dei token di acquisire una comprensione del contesto globale del documento, rendendole sensibili alle informazioni rilevanti per la generazione di sommari. Infine, le rappresentazioni dei segmenti vengono inizializzate mediante l'uso di pooling delle rappresentazioni dei token. Ci sono due approcci principali per calcolare i pesi di pooling. Nel primo metodo si utilizza una media dei pesi di pooling (AvgPool), mentre nel secondo metodo si assegnano pesi adattivi ai token in base alla loro importanza (AdaPool), che può essere appresa da un modello. Questo secondo metodo è più complesso ma può migliorare le prestazioni del modello. Una volta che le rappresentazioni dei token sono state elaborate con successo tramite il processo di "top-down inference", queste rappresentazioni vengono utilizzate da un decoder per generare il sommario finale. Il decoder è progettato come un modello di generazione di sequenze basato su Transformer e utilizza le rappresentazioni dei token per generare frasi sintetiche che costituiranno il sommario del documento.

Tuttavia, ci sono sfide persistenti, come la gestione della lunghezza dei riassunti, la coerenza tematica e la capacità di riconoscere le informazioni più rilevanti. Il campo continua a evolversi, con nuove architetture dei modelli e nuove metriche di valutazione. Complessivamente la text summarization rimane un'area di ricerca vitale nell'NLP, con applicazioni in una vasta gamma di settori, dalla ricerca accademica al giornalismo e all'elaborazione automatizzata di documenti.

Model	ROUGE-1	ROUGE-2	ROUGE-L
PGN	39.53	17.28	37.98
DCA	41.69	19.47	37.92
LEAD	40.42	17.62	36.67
REFRESH	41.0	18.8	37.7
NEUSUM	41.59	19.01	37.98
Transformer$_{BASE}$	38.27	15.03	35.48
PEGASUS$_{BASE}$	41.79	18.81	38.93
PEGASUS$_{LARGE}$ (C4)	43.90	21.20	40.76
PEGASUS$_{LARGE}$ (HugeNews)	**44.17**	**21.47**	**41.11**
BIGBIRD-Pegasus	43.84	21.11	40.74
BART	44.16	21.28	40.90
BERTSUM + Classifier	43.23	20.22	39.60
BERTSUM + Transformer	**43.25**	**20.24**	**39.63**
BERTSUM + LSTM	43.22	20.17	39.59
EASE Token-level sparsity 0.5	43.96	20.91	40.74
EASE Sentence-level sparsity 0.5	43.98	20.95	40.78
Top Down Transformer (AvgPool)	44.32	21.03	41.40
Top Down Transformer (AdaPool)	44.85	21.31	41.15

Tabella 2.1: In questa tabella vengono mostrate le performance di alcuni modelli sopra citati sul dataset CNN/DailyMail.

Model	ROUGE-1	ROUGE-2	ROUGE-L
Transformer$_{BASE}$	35.63	7.95	20.00
PEGASUS$_{BASE}$	34.81	10.16	22.50
PEGASUS$_{LARGE}$ (C4)	44.70	17.27	25.80
PEGASUS$_{LARGE}$ (HugeNews)	44.67	17.18	25.73
BIGBIRD-Pegasus	46.63	19.02	41.77
DANCER + LSTM	41.87	15.92	37.61
DANCER + RUM	42.7	16.54	38.44
DANCER + PEGASUS	45.01	17.60	40.56
LED-large-4,096	44.40	17.94	39.76
LED-large-16,384	46.63	19.62	41.83
Top Down Transformer (AvgPool) (464M)	48.67	20.70	43.91
Top Down Transformer (AdaPool) (464M)	**50.95**	**21.93**	**45.61**

Tabella 2.2: In questa tabella vengono mostrati i principali modelli che hanno ottenuto lo stato dell'arte all'uscita del proprio paper di riferimento, sul dataset arXiv.

Model	ROUGE-1	ROUGE-2	ROUGE-L
BART	45.14	22.27	37.25
BERTSUM	43.85	20.34	39.90
EASE Token-level sparsity 0.5	42.70	19.38	33.81
EASE Sentence-level sparsity 0.5	43.98	20.95	40.78
Transformer$_{BASE}$	38.27	15.03	35.48
PEGASUS$_{BASE}$	41.79	18.81	38.93
PEGASUS$_{LARGE}$ (C4)	43.90	21.20	40.76
PEGASUS$_{LARGE}$ (HugeNews)	44.17	21.47	41.11
BIGBIRD-Pegasus	47.12	24.05	38.80

Tabella 2.3: In questa tabella si confrontano le performance dei modelli sul dataset XSum.

Capitolo 3

Selezione di Frammenti Rilevanti

Fornire ad un language model le parti più significative di un testo può spesso portare a riassunti più accurati e concisi. Tale approccio si basa sul principio del riepilogo estrattivo, in cui le frasi chiave o frammenti di esse sono presenti nel testo originale, vengono selezionate e organizzate per creare un riassunto, come spiegato nel capitolo 2.

Di seguito un elenco dei principali motivi per cui questo approccio può essere più efficace:

- Concentrarsi sulle informazioni importanti: il riepilogo estrattivo dà priorità all'estrazione di frasi o passaggi che contengono le informazioni più critiche, aiutando a produrre un riassunto che evidenzi i punti principali e i dettagli chiave del testo.

- Rumore ridotto: molti testi contengono informazioni ridondanti o meno rilevanti, selezionando solo le parti più significative è possibile ridurre il rumore e rendere il riepilogo più chiaro e conciso.

- Preservare il contesto: aumenta le possibilità di mantenere il contesto originale del testo utilizzando frasi o frammenti esistenti, ciò può rendere il riepilogo più coerente e comprensibile.

- Riduzione del rischio di generazione di false informazioni: quando si generano riassunti, c'è sempre il rischio di travisare le intenzioni dell'autore, usando come base di partenza le frasi o frammenti di frasi comuni minimizza questo rischio perché parafrasa direttamente le frasi del testo originale.

- Coerenza: può aiutare a mantenere lo stile di scrittura e il tono del testo originale, rendendo il riassunto più coerente con il materiale originale.

Tuttavia, è importante notare che, sebbene questo approccio abbia i suoi vantaggi, potrebbe non sempre catturare le sfumature del testo originale o generare riepiloghi che si leggono in modo fluido quanto il riepilogo astrattivo (dove il modello genera riepiloghi con parole proprie).

In definitiva, l'efficacia di un riassunto, sia esso estrattivo o astrattivo, dipende da vari fattori, tra cui la qualità del modello linguistico, la complessità del testo di partenza e il pubblico a cui è rivolto il riassunto.

La tecnica dei frammenti comuni è descritta coi seguenti passi:

1. In primo luogo, avviene l'estrazione dei frammenti (cioè una sequenza di caratteri comuni) tra riassunto e articolo, e viene poi creato un dataset aggiungendo una colonna per tali frammenti

2. In secondo luogo, si valuta a partire da questi frammenti se è possibile generare il riassunto, tramite le metriche ROUGE-1, ROUGE-2 e ROUGE-L. La valutazione avviene eseguendo il fine-tuning di un modello già pre-addestrato, specializzandolo nel generare il riassunto dati i frammenti precedenti.

3. In terzo luogo, se le metriche raggiunte sono maggiori del semplice fine tuning, di un Language Model già pre-addestrato, con articolo in input e in output il riassunto, allora vuol dire che abbiamo ottenuto un possibile incremento delle performance.

4. Per effettivamente notare incrementi notevoli di performance non basta ottenere score alti al punto precedente, poiché tali frammenti sono stati ottenuti in maniera algoritmica, quindi esatta, e quindi necessitano di aver già il riassunto, cosa che ovviamente non siamo già in possesso. Di conseguenza, bisogna prima eseguire un fine-tuning un modello per generare frammenti, e poi fare un fine-tuning di un altro modello sull'output del precedente, ottenendo così un modello specializzato nel generare frammenti dato un testo, e un modello specializzato nel generare un riassunto dati i frammenti generati dal precedente. Ovviamente gli score scenderanno del secondo modello rispetto al passo precedente, e in ogni caso il collo di bottiglia di questa soluzione rimane il minimo score dei due modelli.

3.1 Algoritmo di estrazione di frammenti

L'algoritmo scelto per generare i frammenti è il seguente. Per ogni carattere del riassunto prende la sotto stringa in comune all'articolo più lunga possibile,

poi ricomincia a cercare la prossima sotto stringa, saltando tutti i caratteri su cui ha iterato precedentemente sul riassunto.

```
self._matches =[]
a_start =b_start =0

while a_start <len(a):

    best_match =None
    best_match_length =0

    while b_start <len(b):

        if a[a_start] ==b[b_start]:

            a_end =a_start
            b_end =b_start

            while a_end <len(a) and b_end <len(b) \
                    and b[b_end] ==a[a_end]:

                b_end +=1
                a_end +=1

            length =a_end -a_start

            if length >best_match_length:

                best_match =Fragments.Match(a_start, b_start, length)
                best_match_length =length

            b_start =b_end

        else:

            b_start +=1

    b_start =0

    if best_match:

        if best_match_length >0:
            self._matches.append(best_match)

        a_start +=best_match_length

    else:

        a_start +=1
```

Lo snippet di codice è stato preso a questo link (https://github.com/lil-lab/newsroom/blob/master/newsroom/analyze/fragments.py#L227) con il relativo paper di riferimento [Grusky et al., 2020].

Il primo problema che è possibile notare è che l'algoritmo non restituisce le parole ma solo sotto sequenze di caratteri comuni, quindi più difficili da interpretare dal language model. Di seguito una prima soluzione proposta:

```
def from_fragment_to_word(f_start, f_end, string):

    while f_start -1 >0 and string[f_start -1] !=" ":
        f_start -=1

    while f_end +1 <len(string) and string[f_end +1] !=" ":
        f_end +=1

    return f_start, f_end

def match_texts(a, b):

    matches =[]
    a_start =0
    b_start =0

    while a_start <len(a):

        best_match =None

        best_match_length =0

        while b_start <len(b):

            if a[a_start] ==b[b_start]:

                a_end =a_start
                b_end =b_start

                while a_end <len(a) and b_end <len(b) and b[b_end] ==
                                                        a[a_end]:
                    b_end +=1
                    a_end +=1

                length =a_end -a_start

                if length >best_match_length:

                    # Se e' un bestmatch secondo l'algoritmo
                    # estrai le parole derivanti da tali lettere
```

```
                a_start_w, a_end_w =from_fragment_to_word(a_start,
                                                          a_end, a)
                b_start_w, b_end_w =from_fragment_to_word(b_start,
                                                          b_end, b)



                # check che le parole siano uguali

                if a[a_start_w:a_end_w] ==b[b_start_w:b_end_w]:
                    # se lo sono ottimo
                    best_match =a[a_start_w:a_end_w]
                    best_match_length =length



                else:
                    # altrimenti skippa
                    b_start +=1
                    continue

            b_start =b_end

        else:

            b_start +=1



    b_start =0

    if best_match:

        if best_match_length >0:
            matches.append(best_match)

        a_start +=best_match_length
    else:
        a_start +=1

coverage =sum(len(o) for o in matches) /len(a)
density =sum(len(o) **2 for o in matches) /len(a)
return coverage, density, matches
```

Invece di prendere come parte comune la sequenza di caratteri, l'algoritmo prende solo le parole comuni, considerando come parole le sequenze di lettere separate da spazi.

3.1.1 Similarità con il riassunto

Tutti gli algoritmi presentati precedentemente hanno un ulteriore problematica, cioè l'altissima similarità con il riassunto. Ciò è un male perché seguendo la tecnica proposta per migliorare i language model, sarebbe quasi inutile dato che il primo modello si specializzerebbe a generare i frammenti che sono già praticamente uguali al riassunto (si tratta di veramente pochi caratteri per ogni coppia riassunto-articolo dei principali dataset di text summarization). Ciò è dovuto dal fatto che l'algoritmo non tiene conto della posizione di tali frammenti di caratteri. Difatti basta solo che la sequenza di caratteri del riassunto sia presente nell'articolo, in qualsiasi posizione o contesto completamente diverso, per essere presa, di conseguenza si può adottare un'ulteriore soluzione.

```
def only_long_fragment(fragments, num_of_spaces=1) ->List[str]:

    res = map(lambda x: x.strip(), fragments)
    res = filter(lambda x: x.count(" ") >=num_of_spaces, res)

    return list(res)
```

In questa funzione bisogna passare in input tutti i frammenti estratti (lista di stringhe) ottenuti con uno degli algoritmi precedenti, e il numero di spazi che tale stringa deve avere. Quindi questa funzione filtra i frammenti passati in input e restituisce i frammenti che hanno solo il numero di spazi richiesti (parametro num_of_spaces di default 1) .

3.1.2 Ottimizzazione estrazione

Questi algoritmi di generazione dei frammenti però hanno grossi problemi di performance. Ciò e dovuto al fatto che siano composti da due loop annidati: il più esterno esegue per tutti i caratteri presenti nel riassunto e il più interno per tutti i caratteri presenti nell'articolo: viene confrontata la sotto sequenza di caratteri in comune, più lunga, iterando tutti i caratteri dell'articolo, ricominciando poi dal riassunto, saltando i caratteri comuni iterati precedentemente, così fino alla fine del ciclo più esterno.

Questi algoritmi hanno una complessità computazionale $\Theta(A \cdot S)$ indicando con S è la lunghezza in caratteri del summary e indicando con A è la lunghezza in caratteri dell'articolo. Tale algoritmo in casi particolari come: la lunghezza del riassunto limitata, oppure la lunghezza dell'articolo è limitata, può anche avere performance accettabili. Lato pratico, questi casi sono più rari, infatti sul dataset di arXiv per calcolare i frammenti di 15,000 articoli, impiega circa 2 ore e 30 minuti.

Si possono implementare principalmente 2 tecniche per ottimizzare questi algoritmi.

Prefix tree o Trie

L' albero dei prefissi, chiamato anche trie (da "retrieval"), è una struttura dati di tipo albero utilizzata per organizzare e memorizzare una grande quantità di dati basati su stringhe in maniera altamente efficiente. Data la sua alta efficienza per le operazioni di ricerca e inserimento, essa è ampiamente utilizzata in molte applicazioni, tra cui motori di ricerca, algoritmi di controllo della grammatica, sistemi di predizione di testo, e molto altro. La sua caratteristica principale è la capacità di recuperare velocemente insiemi di stringhe, come parole, frasi o qualsiasi sequenza di caratteri, in modo efficiente. Di seguito una descrizione delle caratteristiche chiave di un albero dei prefissi:

- Struttura ad albero: Un albero dei prefissi è una struttura dati gerarchica, ma con un numero variabile di figli per ogni nodo. Ogni nodo rappresenta un carattere, per delineare una parola invece si una un flag sul nodo raggiunto, dopo aver esplorato nodo per nodo l'albero. Possiamo notare visivamente la rappresentazione in memoria della struttura dati in Figura 3.1.

- Rappresentazione delle parole: Un albero dei prefissi è ideale per rappresentare un insieme di parole in cui le parole condividono comunemente prefissi. Questo rende l'albero estremamente efficiente nel risparmiare spazio, poiché i prefissi comuni rimangono condivisi tra le parole. Inoltre, riduce notevolmente l'uso di memoria rispetto ad altre strutture dati come le tabelle hash.

- Inserimento e ricerca veloci: L'inserimento di una nuova parola o la ricerca di una parola esistente in un albero dei prefissi sono operazioni estremamente efficienti, poiché il percorso verso il nodo finale rappresentante la parola può essere trovato in tempo $\Theta(W)$, dove W è la lunghezza della parola stessa.

- Supporto per suggerimenti e completamento automatico: L'albero dei prefissi può implementare funzionalità di suggerimento e completamento automatico. Un esempio, quando si inizia a digitare una query in un motore di ricerca, l'albero dei prefissi può essere utilizzato per: generare suggerimenti in tempo reale basati sulle stringhe che iniziano con i caratteri inseriti, compiti come la correzione ortografica, la classificazione di testo, l'analisi dei dati.

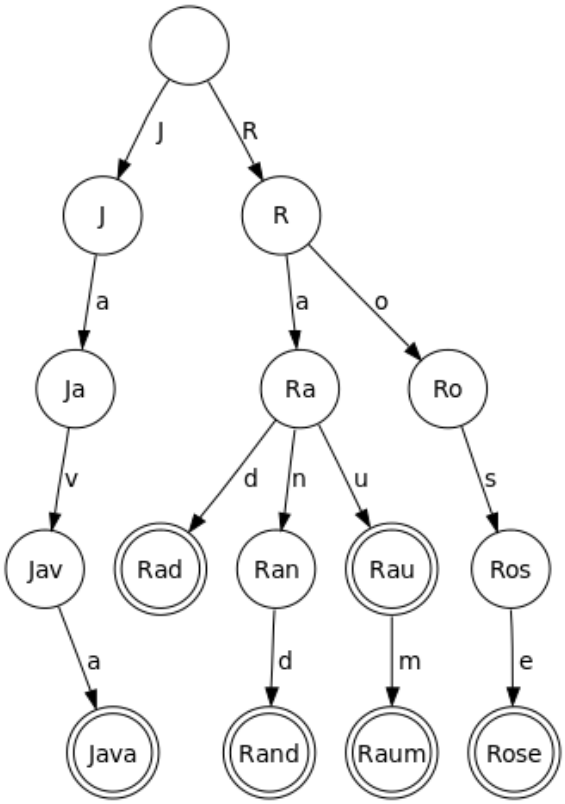

Figura 3.1: La figura mostra la rappresentazione in memoria dell'albero dei prefissi. Come è possibile notare i nodi con 2 cerchi indicano che il percorso fino ad essi ha portato ad una parola, quindi: Java, Rad, Rand, Rau, Raum, Rose. Se nella esplorazione ci si ferma prima significa che la sequenza di caratteri non è una parola.

Di seguito verranno esposte le principali operazioni, già modificate per essere utilizzate per l'ottimizzazione dell'estrazione dei frammenti, e la rappresentazione dei nodi.

- Rappresentazione di un nodo. Children è un dizionario che associa ad un carattere il prossimo **TrieNode**, mentre il flag is_end_of_word verrà messo a True quando verrà inserita una parola che finisce su questo nodo.

```
class TrieNode:
    def __init__(self):
        self.children ={}
        self.is_end_of_word =False
```

- Rappresentazione della struttura dati albero. Questa struttura mantiene il riferimento al nodo **root** per eseguire sempre le operazioni.

```
class Trie:
    def __init__(self):
        self.root =TrieNode()
```

- Inserimento di una parola all'interno della struttura dati. Come possiamo notare, la funzione non restituisce nulla, e possiamo notare che l'ultima istruzione è il settare il flag "is_end_of_word" a True dell'ultimo nodo raggiunto. Il funzionamento di questa operazione è semplice: per ogni carattere della stringa passata in input, vado al nodo del carattere che sto tenendo in considerazione e se per il prossimo carattere il nodo corrente non ha un figlio lo creo, così fino alla fine della stringa. Il costo computazionale di tale operazione è $\Theta(W)$ dove W è il numero di caratteri della parola passata in input.

```
def insert(self, word :str) ->None:
    node =self.root
    for char in word:
        if char not in node.children:
            node.children[char] =TrieNode()
        node =node.children[char]
    node.is_end_of_word =True
```

- Ricerca di una parola. Questa funzione permette di cercare la parola passata in input, in tempo $\Theta(W)$ dove W è il numero di caratteri nella parola. La funzione semplicemente percorre ogni nodo corrispondete alla sequenza di caratteri della parola, se durante il percorso non trova nessun figlio per un carattere significa che tale parola non è presente. Invece una volta finiti i caratteri della parola, bisogna vedere se tale nodo è la fine di una parola mai inserita, quello sarà il valore di ritorno della funzione (il flag "is_end_of_word").

```
def search(self, word :str) ->bool:
    node =self.root
    for char in word:
        if char not in node.children:
            return False
        node =node.children[char]
    return node.is_end_of_word
```

- La seguente funzione è stata modificata dall'originale. Di norma questa funzione viene usata per restituire tutte le parole che iniziano per il

prefisso passato in input. Questa versione invece restituisce True solo se il prefisso passato è presente nella struttura dati. Questa modifica è dovuta dal fatto che nella generazione dei frammenti non ci interessa la parole che possiedono tale prefisso ma se quel prefisso esiste o meno nell'articolo. La complessità computazionale rimane sempre $\Theta(W)$ dove W è il numero di caratteri del prefisso passato.

```
def starts_with(self, prefix :str) ->bool:
    node =self.root

    for char in prefix:
        if char not in node.children:
                return False

        node =node.children[char]
        return True
```

Avendo ora implementato una struttura dati che abbatte la complessità computazionale alla sola lunghezza della parola che ci interessa cercare e inserire, in un testo, bisogna adattare tale struttura dati al problema che stiamo affrontando, cioè trovare i frammenti comuni tra articolo e riassunto. Dato che il testo più lungo è quello dell'articolo, abbattiamo la complessità computazionale che esso comporta; quindi, popoliamo tale struttura dati sulle parole presenti nell'articolo, avendo quindi una complessità computazionale di inserimento delle parole di $\Theta(A)$ dove A è il numero di caratteri presenti nell'articolo. Successivamente si passerà alla ricerca dei frammenti comuni usando questa struttura dati; quindi, per ogni sequenza di caratteri del riassunto se è presente nell'articolo la concateniamo con uno spazio rispetto alle sequenze prese in precedenza. Quindi la complessità computazionale finale sarà la seguente $\Omega(A + S \cdot max(S_i))$, indicando con S la lunghezza in caratteri del summary e indicando con A la lunghezza in caratteri dell'articolo e $max(S_i)$ è la sottosequenza di caratteri comuni più lunga nel riassunto, e quindi notevolmente più bassa. Di seguito l'algoritmo che sfrutta tale struttura dati.

```
def match_texts_with_trie(summary :str, article: str) ->List[str]:
    trie =Trie()

    raw =article.split(" ")
    words =set(map(lambda x: x.strip(), raw))

    for w in words:
        trie.insert(w)

    matches =[]
```

```
    a_start =0
    while a_start <len(summary):

        # scorri in a finche' viene trovato qualcosa nel summary (quindi
                                                nel trie)
        # se a e' uno spazio non fare la ricerca nel trie (ovviamente
                                                perche' non vengono inseriti)

        a_end =a_start +1
        while a_end <len(summary) -1 \
            and trie.starts_with(summary[a_start:a_end]):
            a_end +=1


        if a_end -(a_start +1) >1:
            a_end -=1
            matches.append(summary[a_start:a_end])
            a_start =a_end
        else:
            a_start +=1

    return matches
```

In conclusione, con questa soluzione si è ottenuto uno speedup di circa 10 volte poiché per l'estrazione dei frammenti con questo algoritmo su un processore AMD Ryzen 3900X, impiega circa 12 minuti e 40 secondi contro le 2 ore e 30 minuti. Ovviamente lo scopo dell'elaborato non era velocizzare, ma, nonostante ciò, si possono implementare questi algoritmi e strutture dati per risparmiare molto tempo in caso si decidesse di cambiare politica di estrazione di frammenti.

Rolling hash

Il hash rolling, noto anche come hash scorrevole, è un algoritmo che viene utilizzato principalmente per la corrispondenza efficiente di sottostringhe. L'idea principale di un hash rolling è calcolare un valore hash per una finestra scorrevole di lunghezza fissa mentre si sposta attraverso un set di dati più ampio, consentendo aggiornamenti in tempo costante al valore hash man mano che la finestra si sposta, rendendolo utile in vari algoritmi, tra cui la deduplicazione dei dati, la ricerca di modelli e l'indicizzazione dei testi.

Ecco i principali passi di come funziona tipicamente un hash rolling:

1. L' inizializzazione avviene calcolando il valore hash per la prima finestra di caratteri dell'articolo. Spesso viene fatto utilizzando un algoritmo di hash semplice, come l'hash polinomiale rolling o il controllo di ridondanza

ciclica (CRC). Il valore hash iniziale rappresenta i caratteri all'interno della finestra lunga quanto la sottostringa che stiamo cercando.

2. Scorrimento, se il valore hash precedentemente calcolato è diverso dalla sequenza di caratteri viene fatta scorrere la finestra di un carattere a destra, necessitando però di aggiornare il valore hash per riflettere i contenuti della nuova finestra. Tale aggiornamento deve essere fatto in tempo costante, quindi indipendentemente dalla dimensione della finestra.

3. Aggiornamento del valore hash, per far sì che un rolling hash sia efficiente, deve avere la capacità di aggiornare il valore hash in tempo costante. Ciò significa che l'aggiunta e la rimozione di un carattere dalla finestra non deve richiedere il ricalcolo dell'intero hash da zero di tutta la finestra. Invece, deve essere basato sul valore hash precedente e sul carattere che entra e sul carattere esce dalla finestra.

4. Comparazione tra i due hash, ora che abbiamo il nuovo valore hash aggiornato sulla stringa dell'articolo lo compariamo con l'hash della stringa del riassunto che stiamo cercando. Se sono uguali allora compariamo le due stringhe. Se le due stringhe sono uguali allora abbiamo trovato una corrispondenza, altrimenti abbiamo fatto una comparazione inutile e dobbiamo ripartire dallo scorrimento. Per migliorare le performance si può trovare un algoritmo che minimizzi il numero di collisioni dell'hash, così da ridurre al minimo la comparazione tra stringhe.

Di seguito sono elencate le principali applicazioni:

- Ricerca di Sottostringhe: il rolling hash è comunemente utilizzato per ricercare in modo efficiente una sottostringa all'interno di un testo più grande (come nel caso della ricerca comune dei frammenti, precedentemente discussa).

- Deduplicazione dei Dati: il rolling hash aiuta a identificare chunk duplicati di dati nei sistemi di archiviazione, il che è cruciale per la deduplicazione al fine di risparmiare spazio.

- Analisi dei Pacchetti di Rete: vengono utilizzati nell'analisi dei pacchetti di rete per identificare modelli o firme noti nei flussi di dati.

Di seguito il codice che implementa il rolling hash, usando Robin-Karp:

- Di seguito la funzione che calcola l'hash, di una finestra di caratteri, può essere usata per il pattern che stiamo cercando e dopo aver selezionato la finestra di caratteri iniziali nel testo dell'articolo.

```
def hash_string(s):
    hash_value =0
    for char in s:
        hash_value =(hash_value *256 +ord(char)) % prime
    return hash_value
```

- Di seguito la funzione che permette di aggiornare l'hash, aggiungendo e rimuovendo un carattere.

```
def recompute_hash(prev_hash, removed_char, new_char, pattern_length):
    new_hash =(prev_hash *256 +ord(new_char) -ord(removed_char) *(256
                                        **pattern_length)) % prime
    return new_hash
```

Di seguito invece il codice che fa uso delle due funzioni precedenti per cercare il pattern, passato in input come parametro, nel testo.

```
def rolling_hash_search(text :str, pattern :str, start_position=0) ->int:

    prime =101  # A prime number for hash calculations

    text_length =len(text)
    pattern_length =len(pattern)
    pattern_hash =hash_string(pattern)
    start_window =text[start_position:start_position +pattern_length]
    current_hash =hash_string(start_window)

    for i in range(start_position, text_length -pattern_length +1):
        if current_hash ==pattern_hash and text[i:i +pattern_length] ==
                                                pattern:
            return i # Found a match

        if i <text_length -pattern_length:
            current_hash =recompute_hash(current_hash, text[i], text[i +
                                                    pattern_length],
                                                    pattern_length)

    return -1 # No match found
```

Dopo aver integrato alla funzione di match, la ricerca con rolling hash eseguendo il medesimo test sulle stesse istanze precedentemente utilizzate anche per il prefix tree, non si ottiene lo stesso incremento delle performance, questo perché il rolling hash non abbatte la complessità computazionale dell'algoritmo complessivamente, rimanendo quindi sempre $\Theta(S \cdot A)$, dove A e S sono il numero di caratteri dell'articolo e riassunto, ma ne ottimizza solo una parte, quella di ricerca della sottostringa nell'articolo, rendendola più veloce.

3.1.3 Fine-tuning

Di seguito i risultati statistici del fine-tuning sulla generazione del riassunto dati i frammenti con 1 spazio, con il modello LED-base.

Partition of dataset	ROUGE-1	ROUGE-2	ROUGE-L	BERTSCORE
Train	64.67	48.51	67.44	59.27
Test	63.25	46.27	66.02	57.16
Eval	62.48	45.4	65.3	56.58

Tabella 3.1: In questa tabella vengono mostrati i risultati del fine-tuning da frammenti con minimo uno spazio a riassunto.

Mentre il fine-tuning da articolo a frammenti con 1 spazio.

Partition of dataset	ROUGE-1	ROUGE-2	ROUGE-L	BERTSCORE
Test	38.64	16.16	37.41	33.07
Eval	37.92	15.63	36.71	32.45

Tabella 3.2: In questa tabella vengono mostrati i risultati del fine-tuning da articolo a frammenti con minimo uno spazio.

Come possiamo notare la generazione di questi frammenti fa da collo di bottiglia, abbassando quindi notevolmente la capacità del modello di generare riassunti.

3.2 Algoritmo di estrazione delle frasi

Un altro algoritmo che può essere usato al posto dei frammenti, è quello di estrazione delle frasi, si scelgono le frasi dell'articolo che hanno maggiore similarità con le frasi del riassunto. Successivamente come con i frammenti si esegue il fine-tuning di un Language Model per generare queste frasi, e si esegue il fine-tuning che da queste frasi generi il riassunto. Di seguito l'algoritmo che determina le frasi, dato riassunto e articolo:

1. Si estraggono i frammenti, ma in maniera leggermente differente, dato che nella sezione 3.1 il problema da risolvere è se inserire o meno una determinata sequenza comune, qua il problema si è spostato nell'estrazione della sequenza più comune tra riassunto e articolo. Come nell'algoritmo dei frammenti, si fa uso dell'albero dei prefissi per velocizzare la ricerca sull'articolo. Prima di elencare i passi per l'estrazione di questo algoritmo serve definire una funzione che ci permette di estrarre le informazioni sulle

parti comuni tra articolo e riassunto, come numero di spazi e lunghezza in caratteri della parte comune.

Ora possiamo partire nel descrivere l'algoritmo per estrarre il frammento più lungo in comune tra articolo e riassunto.

(a) Per prima cosa creiamo l'albero dei prefissi sull'articolo salvandoci anche la posizione in cui quella parola è stata incontrata.

(b) Ora per ogni carattere del riassunto estraiamo una parola e cerchiamone le sue posizioni all'interno dell'articolo, e per ogni posizione estraiamo il frammento che ha la corrispondenza più lunga, e aggiungiamo tale frammento alla lista che restituiremo. Inoltre, per diminuire il numero di frasi, e concentrarsi più sulle frasi che possibilmente ci daranno più informazioni per generare il riassunto, si possono escludere le stop word, come prima parola per iniziare il frammento.

2. Per ogni frammento estratto al punto precedente si controlla in maniera efficiente, tramite albero degli intervalli, che tale frammento non cada già in una frase già presa, se invece non è già stata presa, dal frammento con relativa posizione salvata nell'articolo. Di seguito il codice che fa uso di tale struttura dati, albero degli intervalli, già implementato nella libreria **intervaltree** di python.

Di seguito la funzione che ci permette di estrarre le informazioni sulle parti comuni tra articolo e riassunto, come numero di spazi e lunghezza in caratteri della parte comune:

```
def get_common_chars_num(summary, article, s_start, a_start,
                                    only_words=True):
    '''
        Return the number of common chars between summary and article
                                            starting from s_start and
                                            a_start
        and the number of spaces between the two fragments
        used for get the longest common fragment
    '''
    common =0
    spaces =0
    start_s =s_start
    start_a =a_start

    while start_s <len(summary) \
        and start_a <len(article) and summary[start_s] ==article[start_a]:
        spaces +=1 if article[start_a] ==" " else 0
        start_s +=1
```

```
        start_a +=1
        common +=1

    start_s -=1
    start_a -=1

    if only_words and spaces >0 and summary[start_s] !=" ":

        while start_s >=0 and start_a >=0 and summary[start_s] !=" " :
            start_s -=1
            start_a -=1

        spaces -=1

    if summary[start_s] ==" " and article[start_a] ==" ":
        spaces =max(spaces -1,0)

    return common, spaces, start_a
```

Di seguito la costruzione della variante del prefix tree, con le parole presenti nell'articolo.

```
def match_text_long_with_trie(summary, article, min_spaces=1,
                                        only_words=True):
    '''
        Return the fragments by the two options:
            - only_words: only the fragments that are surrounded by spaces
            - min_spaces: the minimum number of spaces inside the fragments
    '''
    trie =TrieWithPositions()

    # initialize the trie with the all the words in the article ( sorrunded
                                            by spaces )
    a_start =0
    a_end =0

    while a_end <len(article) and a_start <len(article):
        if article[a_start] !=" ":

            a_end =a_start +1
            while a_end <len(article) and article[a_end] !=" ":
                a_end +=1

            trie.insert(article[a_start:a_end], a_end)

            a_start =a_end +1

        else:
            a_start +=1
```

Di seguito invece l'uso dell'albero dei prefissi per estrarre le parole dal riassunto e ottenere le sue posizioni all'interno dell'articolo, e per ogni posizione estrarre il frammento che ha la corrispondenza più lunga, e aggiungere tale frammento alla lista che restituiremo.

```
def match_text_long_with_trie(summary, article, min_spaces=1,
                                    only_words=True):
    ...
    # now search the for the matches with the trie
    matches =[]
    summary_start =0

    while summary_start <len(summary):


        if summary[summary_start] !=" ":
            summary_end =summary_start +1

            while summary_end <len(summary) and summary[summary_end] !=" ":
                summary_end +=1

            word =summary[summary_start:summary_end]

            found, positions =trie.search(word, get_positions=True)
            if found:
                res =map(lambda x: get_common_chars_num(summary, article,
                                                        summary_end, x),
                                                        positions)
                res =filter(lambda x: x[1] >=min_spaces, res)
                res =max(res, key=lambda x: x[0], default=None)

                if res:
                    s_len =summary_end +res[0] -summary_start
                    tmp_start =res[2] -s_len +1
                    tmp_end =res[2]
                    matches.append((article[res[2] -s_len +1:res[2]],
                                                        (tmp_start,
                                                        tmp_end)))

            summary_start =summary_end +1

        else:
            summary_start +=1


    return matches
```

Di seguito lo stesso algoritmo precedente che però ignora le stopword, come

parola iniziale del frammento.

```
def match_text_long_with_trie(summary, article, min_spaces=1,
                                    only_words=True):
    ...
    # now search the for the matches with the trie
    matches =[]
    summary_start =0

    while summary_start <len(summary):


        if summary[summary_start] !=" ":
            summary_end =summary_start +1

            while summary_end <len(summary) and summary[summary_end] !=" ":
                summary_end +=1

            word =summary[summary_start:summary_end]

            if not word.strip() in stop_words:
                # ignore the first word if is stopword
                found, positions =trie.search(word, get_positions=True)
                if found:
                    res =map(lambda x: get_common_chars_num(summary,
                                                            article,
                                                            summary_end, x),
                                                            positions)
                    res =filter(lambda x: x[1] >=min_spaces, res)
                    res =max(res, key=lambda x: x[0], default=None)

                    if res:
                        s_len =summary_end +res[0] -summary_start
                        tmp_start =res[2] -s_len +1
                        tmp_end =res[2]
                        matches.append((article[res[2] -s_len +1:res[2]],
                                                            (tmp_start,
                                                            tmp_end)))

            summary_start =summary_end +1

        else:
            summary_start +=1


    return matches
```

Di seguito la funzione che dati i frammenti, ne estrae le frasi, ed evita di prendere le frasi duplicate.

```
def get_phrase_with_trie_interval_tree(fragments_with_info, article :str,
                                       append_infos=False):
    '''
        Get phrase from article using trie, only the phrase that have
                                            common words with summary
    '''
    tree_interval =IntervalTree()
    phrase_greedy_order =[]

    for fragment in fragments_with_info:
        start, end =fragment[1]

        if any(interval for interval in tree_interval[start]):
            continue

        while start >0 and ( article[start] !="." or article[start +1] !="
                                            " ):
            start -=1

        while end <len(article) and ( article[end] !="." or article[end -1]
                                            !=" " ):
            end +=1

        if start >0:
            start +=1

        tree_interval.addi(start, end +1)

        phrase =article[start:end +1]
        phrase =phrase.replace("\n", " ").strip()
        to_append =phrase if not append_infos else (phrase, fragment)
        phrase_greedy_order.append(to_append)

    return phrase_greedy_order
```

3.2.1 Interval tree

L'Interval Tree, o meglio in italiano "albero degli intervalli", è una struttura dati avanzata utilizzata per eseguire delle determinate operazioni su una serie di intervalli. Questa struttura è particolarmente utile quando si devono effettuare operazioni di ricerca e interrogazione sugli intervalli, come ad esempio trovare tutti gli intervalli che si sovrappongono con un punto dato. L'Interval Tree ha la stessa struttura di un albero binario bilanciato che organizza gli intervalli in modo efficiente per consentire operazioni di: ricerca, inserimento, eliminazione in maniera efficiente.

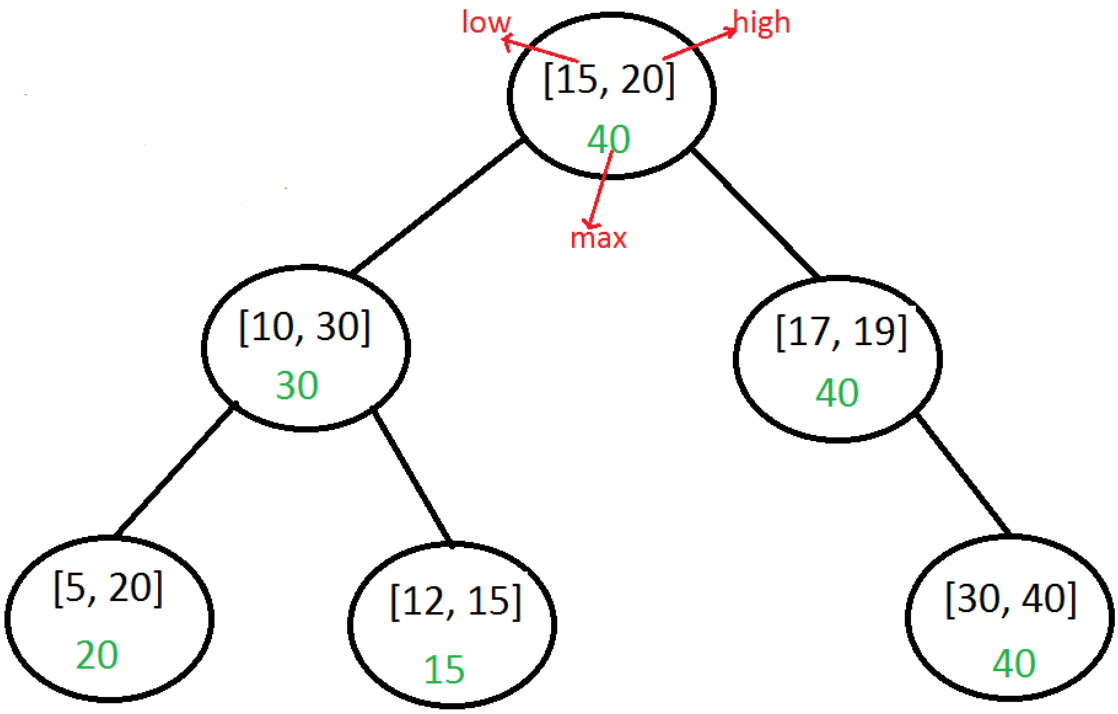

Figura 3.2: In questa immagine viene mostrato la struttura di un interval tree.

L'Interval Tree è strutturato da nodi, ognuno dei quali rappresenta un intervallo. Ogni nodo ha tre componenti principali:

- Intervallo: Rappresenta l'intervallo numerico che il nodo contiene, rappresentato dai due estremi, ad esempio, [x, y], dove "x" è l'estremo inferiore e "y" è l'estremo superiore dell'intervallo.

- Sottoalberi sinistro e destro: Ogni nodo può avere due sottoalberi, uno per gli intervalli che si sovrappongono al nodo a sinistra e uno per gli intervalli che non si sovrappongono a destra. Tale divisione consente di scartare metà del numero di intervalli di ogni sottoalbero da esaminare durante la ricerca.

- Punto Medio dell'Intervallo (midpoint dell'intervallo): Questo valore rappresenta il punto medio dell'intervallo associato a un nodo, e questo valore viene utilizzato per determinare se un intervallo da cercare deve essere esplorato sia a sinistra che a destra del nodo corrente durante la ricerca di intervalli sovrapposti.

Operazioni sull'Interval Tree:

- Inserimento di un Intervallo: L'inserimento di un nuovo intervallo nell'Interval Tree si inizia dalla radice e si scende nell'albero in modo simile a come si farebbe in un albero binario di ricerca. Durante l'inserimento, vengono aggiornati i valori di "midpoint dell'intervallo" lungo il percorso.

- Ricerca di Intervallo sovrapposto:

  1. Inizia dalla radice dell'Interval Tree.
  2. Ci sono tre casi possibili quando confronti l'intervallo del nodo corrente con l'intervallo di ricerca:
     - Se l'intervallo del nodo corrente si sovrappone completamente con l'intervallo di ricerca, significa che tutti gli intervalli nel sotto-albero sinistro del nodo corrente si sovrappongono con l'intervallo di ricerca. Pertanto, puoi aggiungere tutti gli intervalli in questo sotto-albero alla lista dei risultati.
     - Se l'intervallo di ricerca è completamente a sinistra dell'intervallo rappresentato dal nodo corrente: ciò implica che l'intervallo corrispondente al nodo corrente è completamente a destra dell'intervallo di ricerca. In questo caso, non è necessario cercare ulteriormente nel sottoalbero destro di questo nodo, poiché non ci saranno sovrapposizioni.
     - Se l'intervallo di ricerca è completamente a destra dell'intervallo rappresentato dal nodo corrente: ciò implica che l'intervallo corrispondente al nodo corrente è completamente a sinistra dell'intervallo di ricerca. In questo caso, non è necessario cercare ulteriormente nel sottoalbero sinistro di questo nodo, poiché non ci saranno sovrapposizioni.
  3. Ripeti il processo precedente per il sottoalbero sinistro e destro, accumulando gli intervalli che si sovrappongono.

- Ricerca di un Punto in un Intervallo:

  1. Confronta il punto con l'intervallo rappresentato dal nodo corrente.
  2. Ci sono diversi casi possibili quando confronti il punto con l'intervallo del nodo corrente:
     - Se il punto è uguale all'estremo inferiore dell'intervallo o all'estremo superiore dell'intervallo, allora il punto è contenuto in quell'intervallo, allora restituiamo già l'intervallo trovato.

  - Se il punto è strettamente maggiore dell'estremo inferiore dell'intervallo ma strettamente minore dell'estremo superiore dell'intervallo, allora il punto è contenuto all'interno di quell'intervallo, allora si aggiunge quell'intervallo alla lista dei risultati e continuare la ricerca nei sottoalberi sinistro e destro del nodo corrente, se un intervallo non basta.
  - Se il punto è minore dell'estremo inferiore dell'intervallo, significa che dovrai cercare solo nel sottoalbero sinistro del nodo corrente poiché non ci saranno sovrapposizioni a destra.
  - Se il punto è maggiore dell'estremo superiore dell'intervallo, significa che dovrai cercare solo nel sottoalbero destro del nodo corrente poiché non ci saranno sovrapposizioni a sinistra.

3. Ripete il passo precedente fino a che non si hanno esplorato tutte le foglie interessate.

- Eliminazione di un Intervallo: Per eliminare un intervallo dall'Interval Tree, si esegue una ricerca per trovare il nodo corrispondente all'intervallo da eliminare e poi si lo elimina seguendo le regole dell'eliminazione in un albero binario.

L'Interval Tree è una struttura dati potente per la gestione efficiente di intervalli e rappresenta una soluzione efficace per molte applicazioni, come la gestione di orari di appuntamenti, l'analisi di dati temporali o la risoluzione di problemi di sovrapposizione di intervalli. Grazie alla sua organizzazione bilanciata è in grado di ridurre notevolmente la complessità temporale delle operazioni di ricerca e interrogazione sugli intervalli, garantendo prestazioni efficienti anche su grandi dataset di intervalli.

Ciò mi ha permesso di inserire frasi facendo si che non siano già state inserite precedentemente.

3.2.2 Fine-tuning

Di seguito i risultati del fine-tuning del modello LED base nella generazione del riassunto date le frasi. Verranno mostrati risultati per un numero crescente di spazi per il match, poiché per le frasi avendo più rumore, può aver senso ridurle di numero, mentre per i frammenti che sono a contenuto altamente informativo, e l'aumento di numero degli spazi farebbe perdere informazioni essenziali.

Di seguito gli esperimenti eseguiti su SciLay, un dataset che comprende 43.790 istanze, ognuna corrispondente a un articolo scientifico nel campo biomedico. L'obiettivo principale di SciLay è fornire supporto per lo sviluppo

Partition of dataset	ROUGE-1	ROUGE-2	ROUGE-L	BERTSCORE
Eval	48.29	22.38	49.47	44.78
Test	48.83	22.77	50.03	45.25

Tabella 3.3: In questa tabella vengono mostrati i risultati del fine-tuning da frasi con match minimo uno spazio a riassunto.

Partition of dataset	ROUGE-1	ROUGE-2	ROUGE-L	BERTSCORE
Eval	46.77	21.77	48.11	44.08
Test	47.41	22.43	48.98	44.85

Tabella 3.4: In questa tabella vengono mostrati i risultati del fine-tuning da frasi con match minimo 2 spazi a riassunto.

e la valutazione di modelli di riassunto del testo capaci di semplificare in modo efficace il linguaggio scientifico complesso, conservando al contempo le informazioni essenziali. Ciascun articolo è pubblicato su una rivista scientifica, con 15 diverse classificazioni di riviste presenti nel dataset.

Partition of dataset	ROUGE-1	ROUGE-2	ROUGE-L	BERTSCORE
Eval	45.79	21.46	47.43	43.5
Test	45.7	21.4	47.44	43.56

Tabella 3.5: In questa tabella vengono mostrati i risultati del fine-tuning da frasi con match minimo 3 spazi a riassunto.

Partition of dataset	ROUGE-1	ROUGE-2	ROUGE-L	BERTSCORE
Eval	43.62	20.28	45.42	41.05
Test	43.46	20.57	45.49	41.54

Tabella 3.6: In questa tabella vengono mostrati i risultati del fine-tuning da frasi con match minimo 4 spazi a riassunto.

Partition of dataset	ROUGE-1	ROUGE-2	ROUGE-L	BERTSCORE
Eval	41.88	19.64	43.97	40.45
Test	41.73	20.04	44.09	40.8

Tabella 3.7: In questa tabella vengono mostrati i risultati del fine-tuning da frasi con match minimo 5 spazi a riassunto.

Partition of dataset	ROUGE-1	ROUGE-2	ROUGE-L	BERTSCORE
Eval	36.96	14.65	39.42	37.55
Test	36.74	14.34	39.14	37.52

Tabella 3.8: In questa tabella vengono mostrati i risultati del fine-tuning da frasi con match minimo 1 spazo, ignorando le stopword, a riassunto.

Partition of dataset	ROUGE-1	ROUGE-2	ROUGE-L	BERTSCORE
Eval	40.83	16.24	42.37	38.11
Test	40.74	16.08	42.42	38.43

Tabella 3.9: In questa tabella vengono mostrati i risultati del fine-tuning da frasi con match minimo 2 spazi, ignorando le stopword, a riassunto.

Partition of dataset	ROUGE-1	ROUGE-2	ROUGE-L	BERTSCORE
Eval	39.5	18.87	41.63	39.22
Test	39.66	18.7	41.63	40.04

Tabella 3.10: In questa tabella vengono mostrati i risultati del fine-tuning da frasi con match minimo 3 spazi, ignorando le stopword, a riassunto.

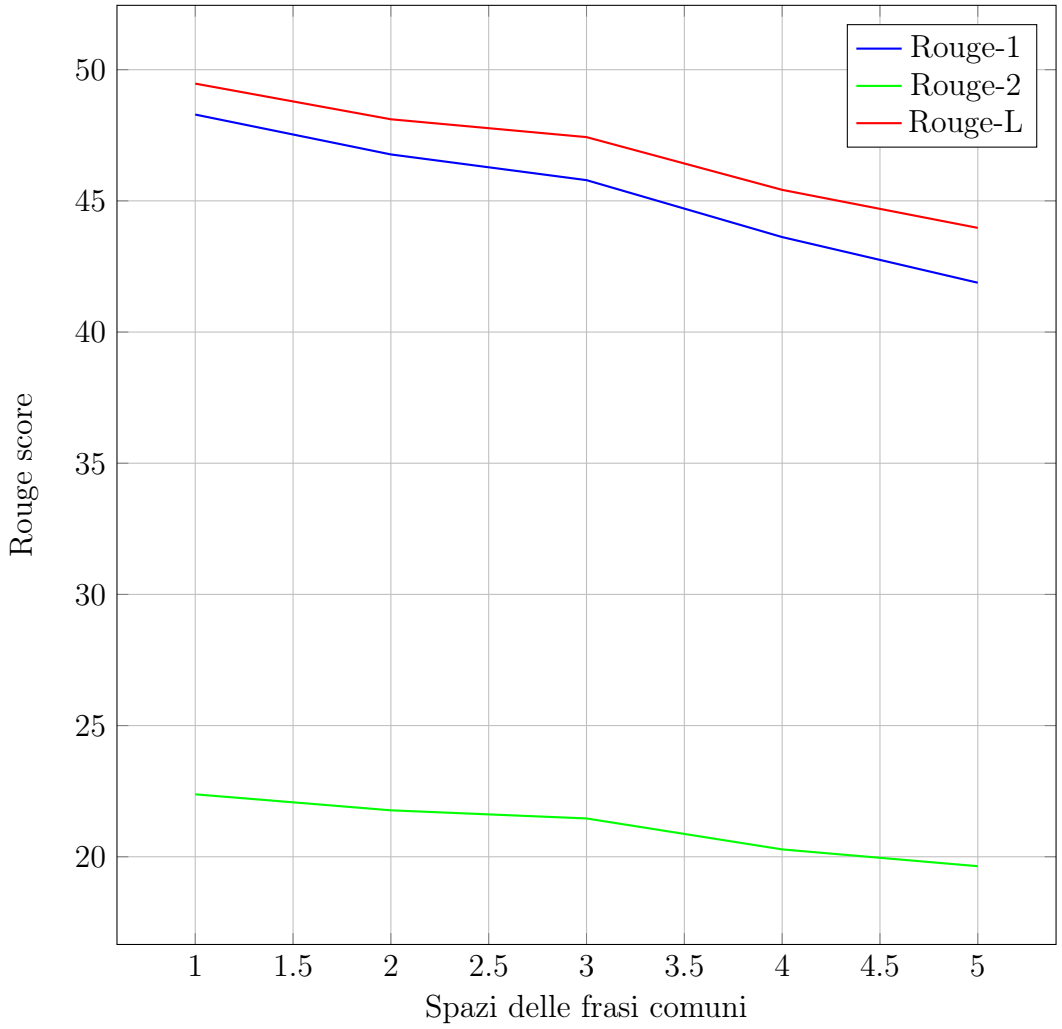

Figura 3.3: Grafico dei punteggi ROUGE del modello LED, in funzione del numero di spazi delle frasi in comune, sul dataset arXiv. In input al modello vengono date le frasi comuni esatte, e da esse si genera il riassunto. Si evidenzia il fatto che aumentando il numero di spazi per le frasi comuni la precisione del riassunto cala.

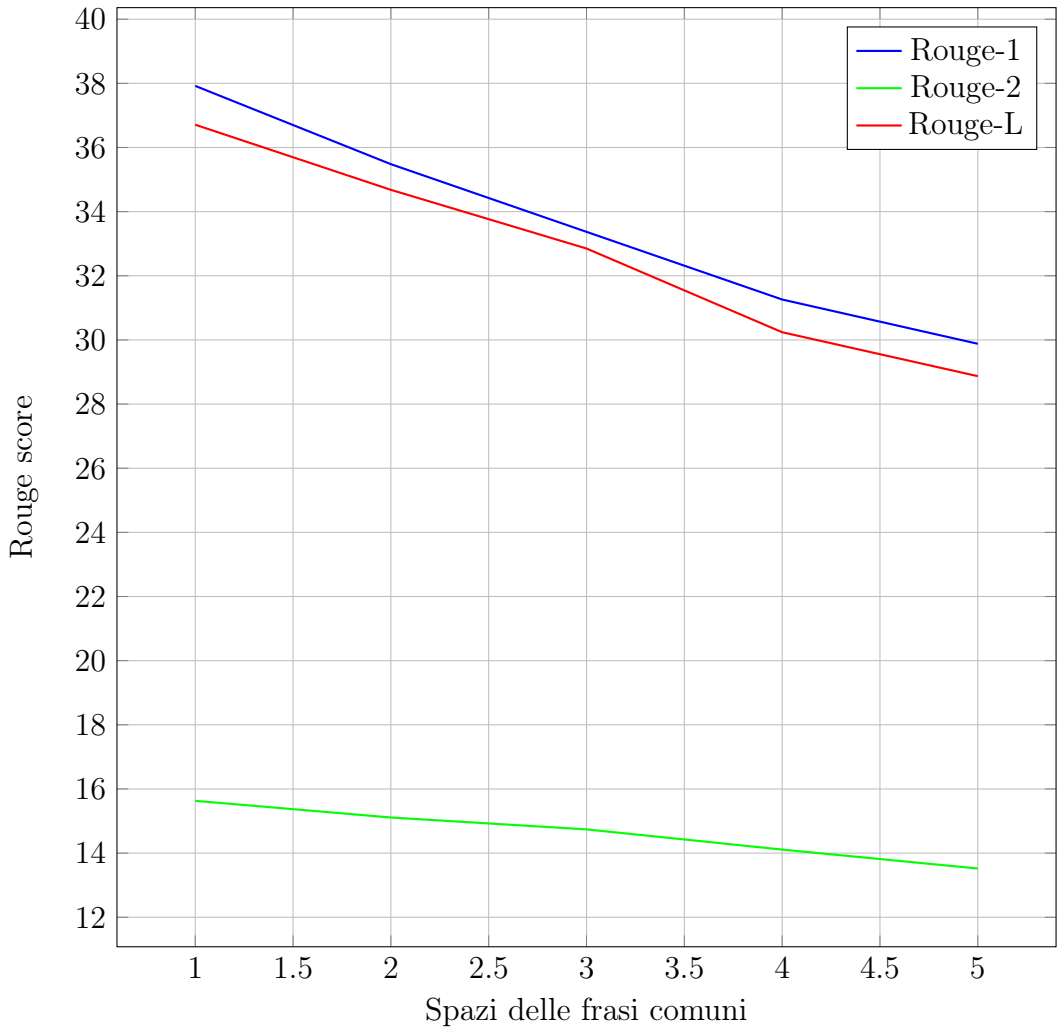

Figura 3.4: Grafico dei ROUGE del modello LED, in funzione del numero di spazi delle frasi in comune, su dataset arXiv. In input al modello vengono dati gli articoli e in output si fornisce le frasi comuni esatte, e da esse si genera il riassunto. Si evidenzia il fatto che aumentando il numero di spazi per le frasi comuni la precisione del riassunto cala.

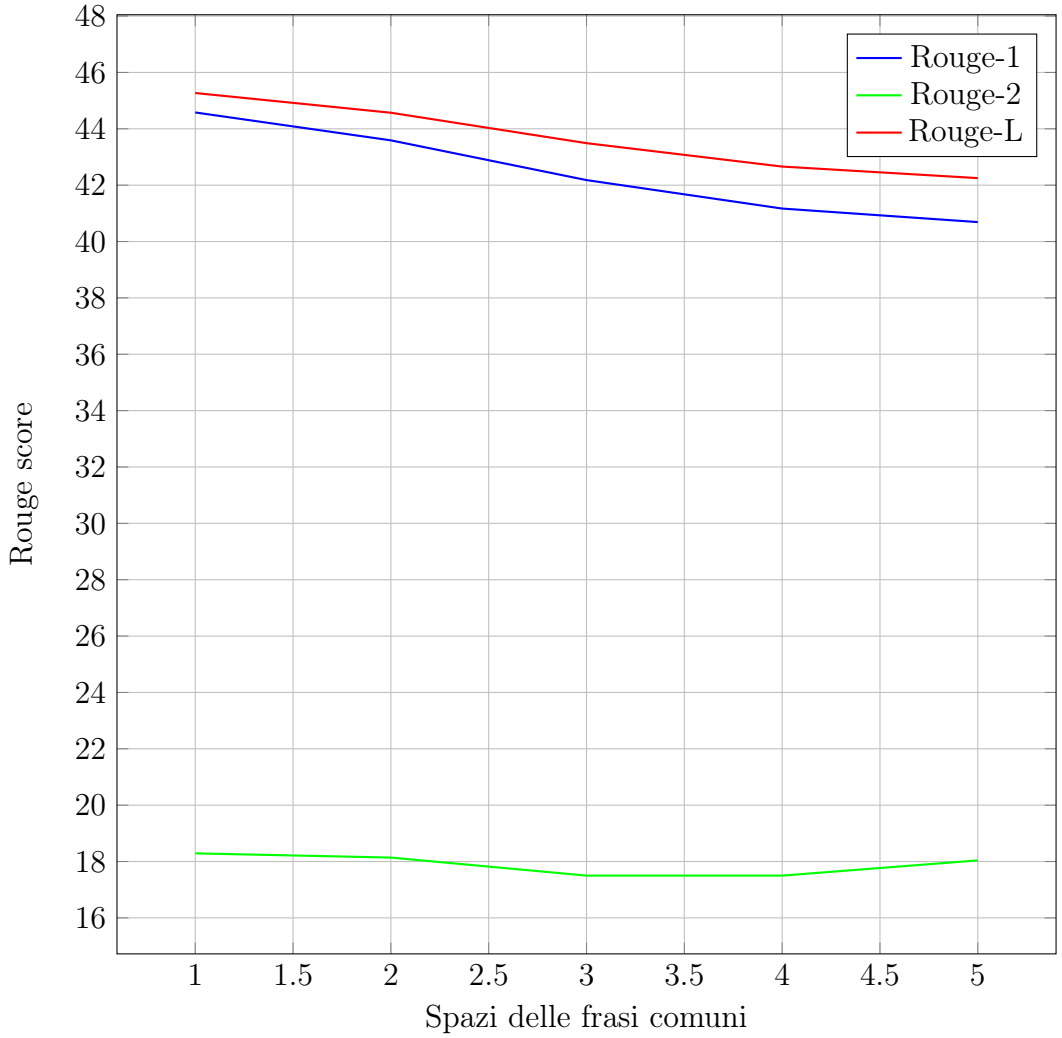

Figura 3.5: Grafico dei punteggi ROUGE del modello Pegasus, in funzione del numero di spazi delle frasi in comune, su dataset arXiv. In input al modello vengono date le frasi comuni esatte, e da esse si genera il riassunto. Si evindezia il fatto che aumentando il numero di spazi per le frasi comuni la precisione del riassunto cala.

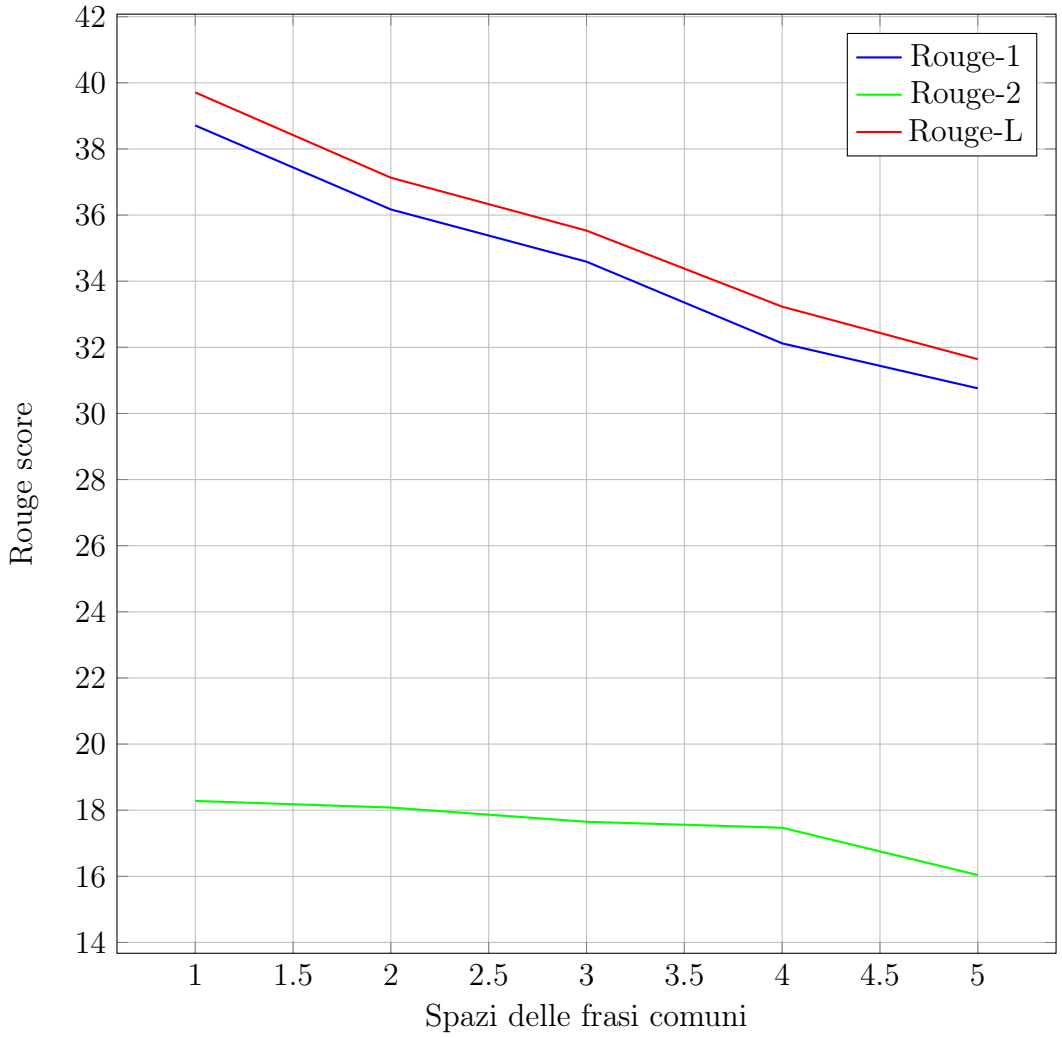

Figura 3.6: Grafico dei punteggi ROUGE del modello Pegasus, in funzione del numero di spazi delle frasi in comune, su dataset arXiv. In input al modello vengono dati gli articoli e l'output fornisce le frasi comuni esatte, e da esse si genera il riassunto. Si evidenzia il fatto che aumentando il numero di spazi per le frasi comuni la precisione del riassunto cala.

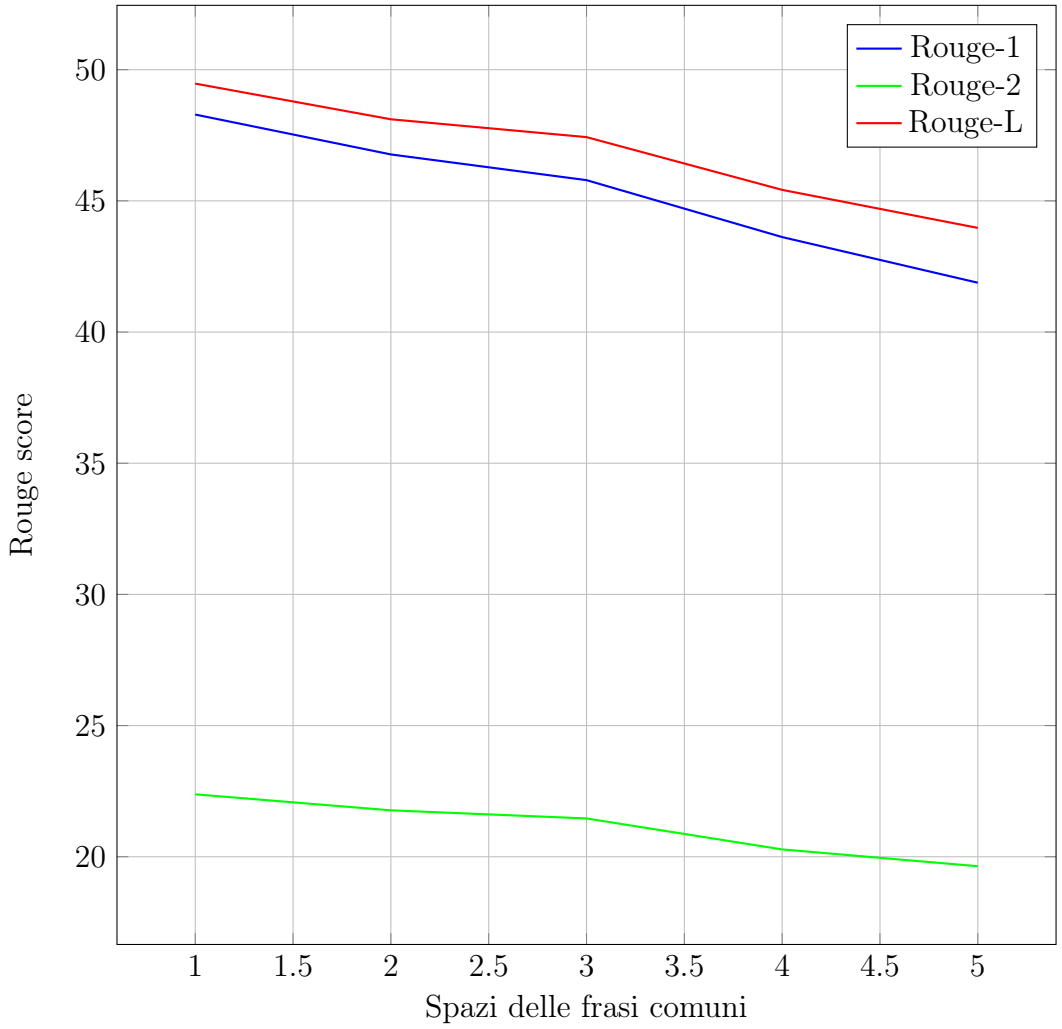

Figura 3.7: Grafico dei punteggi ROUGE del modello LED, in funzione del numero di spazi delle frasi in comune, sul dataset SciLay. In input al modello vengono date le frasi comuni esatte, e da esse si genera il riassunto. Si evidenzia il fatto che aumentando il numero di spazi per le frasi comuni la precisione del riassunto cala.

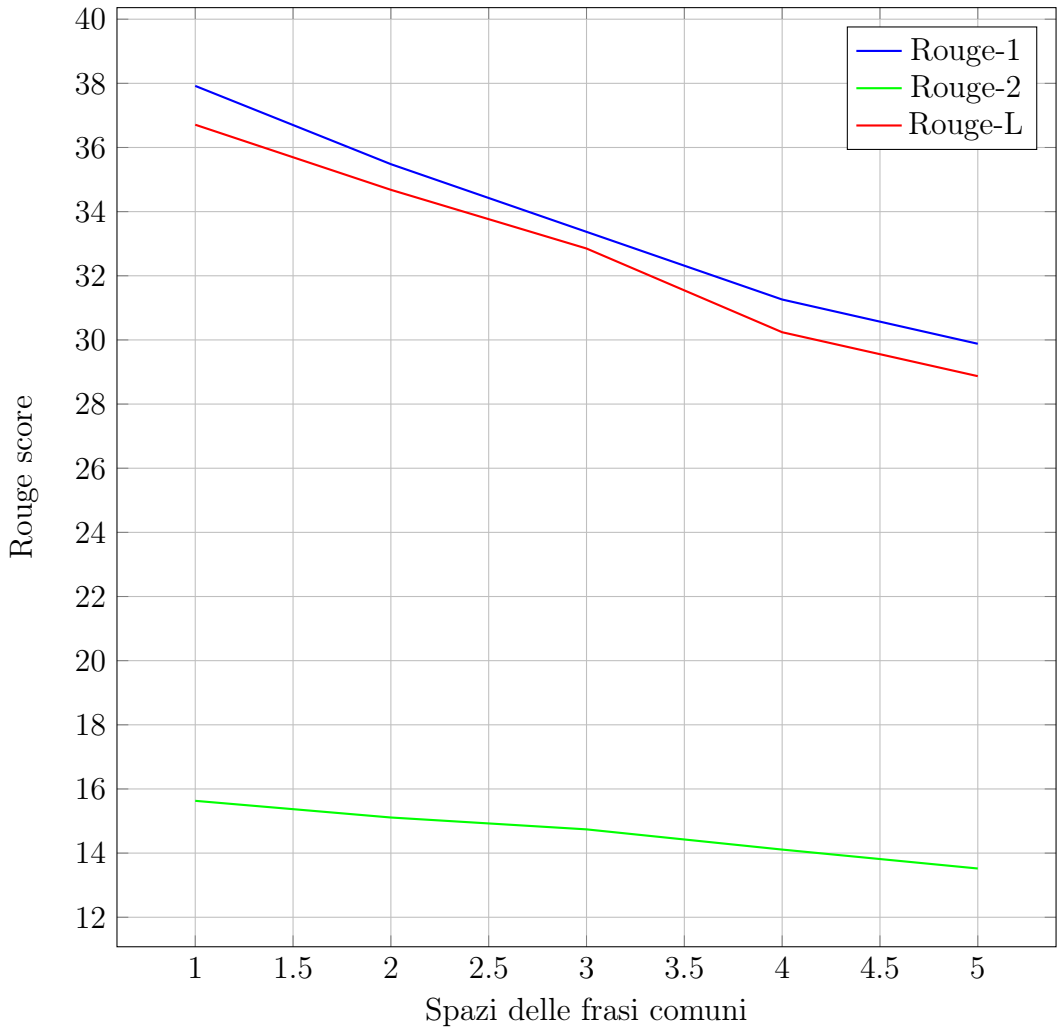

Figura 3.8: Grafico dei punteggi ROUGE del modello LED, in funzione del numero di spazi delle frasi in comune, su dataset SciLay. In input al modello vengono dati gli articoli e l'output fornisce le frasi comuni esatte, e da esse si genera il riassunto. Si evidenzia il fatto che aumentando il numero di spazi per le frasi comuni la precisione del riassunto cala.

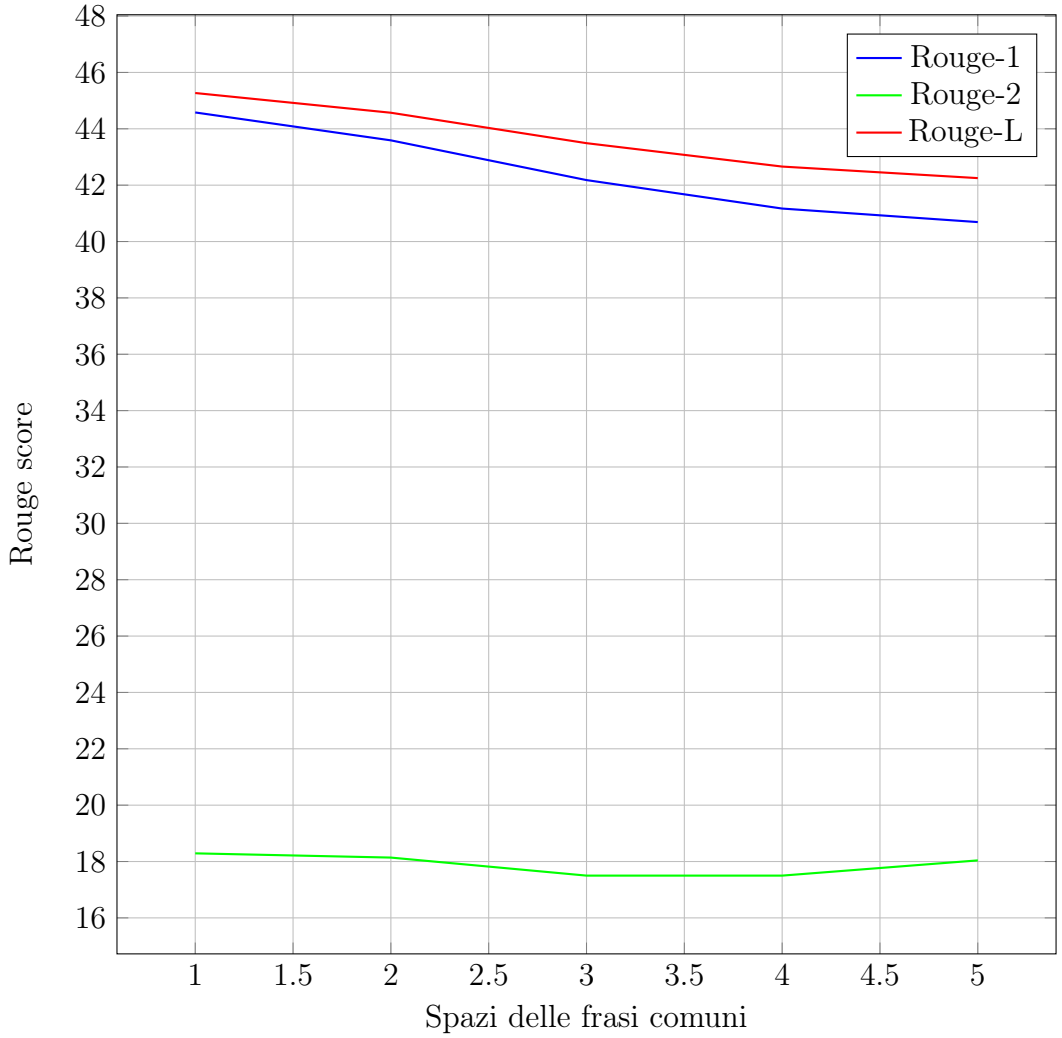

Figura 3.9: Grafico dei punteggi ROUGE del modello Pegasus, in funzione del numero di spazi delle frasi in comune, su dataset SciLay. In input al modello vengono date le frasi comuni esatte, e da esse si genera il riassunto. Si evidenzia il fatto che aumentando il numero di spazi per le frasi comuni la precisione del riassunto cala.

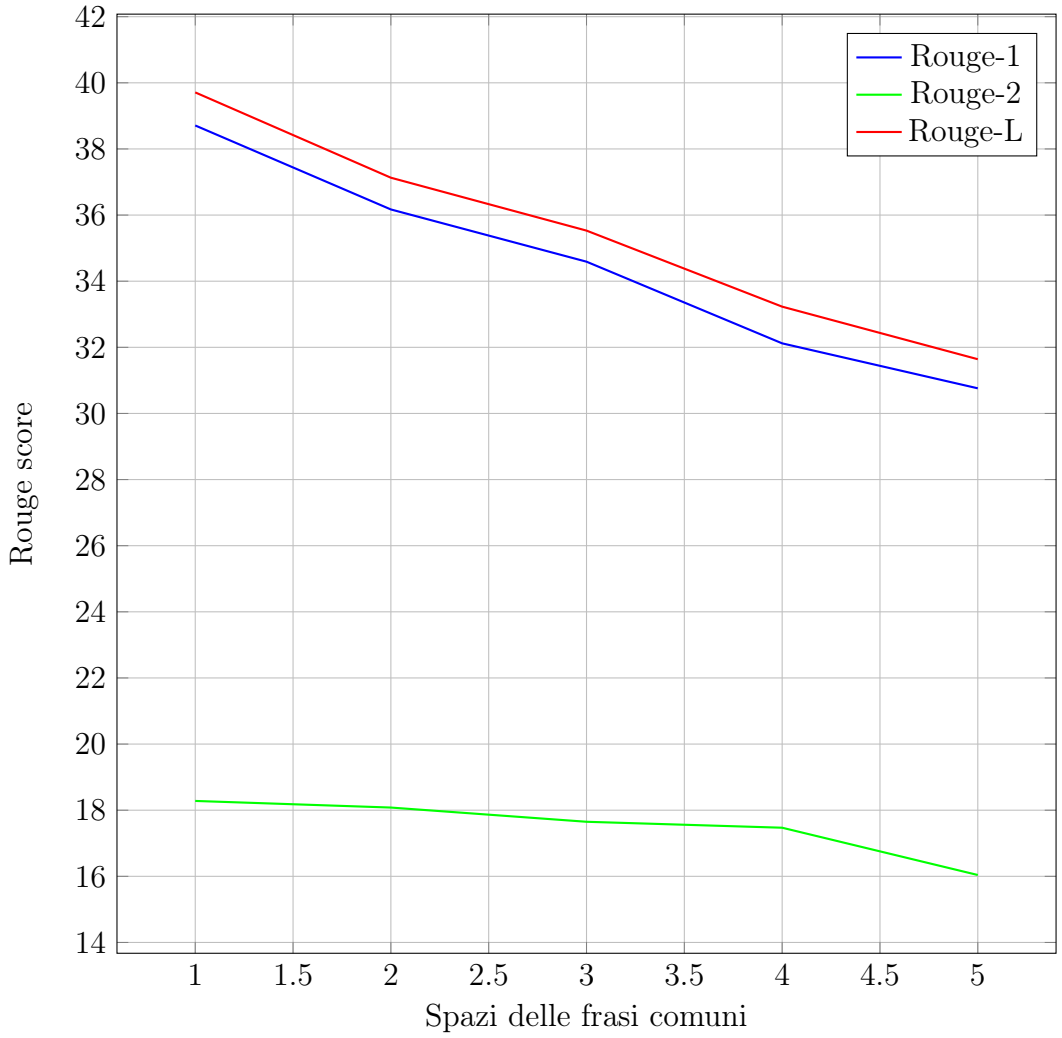

Figura 3.10: Grafico dei punteggi ROUGE del modello Pegasus, in funzione del numero di spazi delle frasi in comune, su dataset SciLay. In input al modello vengono dati gli articoli e in output fornice le frasi comuni esatte, e da esse si genera il riassunto. Si evidenzia il fatto che aumentando il numero di spazi per le frasi comuni la precisione del riassunto cala.

Capitolo 4

Conclusione

In questo lavoro, nel contesto della text summarization, si è riuscito ad evidenziare le limitazioni degli attuali language model nel concentrarsi sulle parti rilevanti di un documento, coinvolgendo la specializzazione di un modello nella generazione di sequenze di testo o frasi comuni tra l'articolo originale e il suo riassunto. Il lavoro evidenzia che ci sono opportunità per miglioramenti significativi in alcuni contesti. Ad esempio, la specializzazione del modello che genera il riassunto da frammenti raggiunge score elevatissimi, mai raggiunti prima, mentre il modello che li genera ottiene score molto più bassi, e quindi un approccio inconcludente. Allo stesso modo, il modello che genera il riassunto a partire da frasi comuni con uno spazio mostra miglioramenti rispetto alla specializzazione diretta. Tuttavia, attualmente, i modelli non sono in grado di generare queste frasi in quanto hanno un limite massimo di token in output (1024), mentre queste frasi sono in media più lunghe.

Bibliografia

[Beltagy et al., 2020] Beltagy, I., Peters, M. E., and Cohan, A. (2020). Longformer: The long-document transformer.

[Celikyilmaz et al., 2018] Celikyilmaz, A., Bosselut, A., He, X., and Choi, Y. (2018). Deep communicating agents for abstractive summarization. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1662–1675, New Orleans, Louisiana. Association for Computational Linguistics.

[Cohan et al., 2018] Cohan, A., Dernoncourt, F., Kim, D. S., Bui, T., Kim, S., Chang, W., and Goharian, N. (2018). A discourse-aware attention model for abstractive summarization of long documents.

[Dong et al., 2019] Dong, L., Yang, N., Wang, W., Wei, F., Liu, X., Wang, Y., Gao, J., Zhou, M., and Hon, H.-W. (2019). Unified language model pre-training for natural language understanding and generation.

[Gidiotis and Tsoumakas, 2020] Gidiotis, A. and Tsoumakas, G. (2020). A divide-and-conquer approach to the summarization of long documents.

[Grusky et al., 2020] Grusky, M., Naaman, M., and Artzi, Y. (2020). Newsroom: A dataset of 1.3 million summaries with diverse extractive strategies.

[Lewis et al., 2019] Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., and Zettlemoyer, L. (2019). Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension.

[Li et al., 2021] Li, H., Einolghozati, A., Iyer, S., Paranjape, B., Mehdad, Y., Gupta, S., and Ghazvininejad, M. (2021). EASE: Extractive-abstractive summarization end-to-end using the information bottleneck principle. In *Proceedings of the Third Workshop on New Frontiers in Summarization*, pages

85–95, Online and in Dominican Republic. Association for Computational Linguistics.

[Liu, 2019] Liu, Y. (2019). Fine-tune bert for extractive summarization.

[Narayan et al., 2018] Narayan, S., Cohen, S. B., and Lapata, M. (2018). Ranking sentences for extractive summarization with reinforcement learning.

[Pang et al., 2022] Pang, B., Nijkamp, E., Kryściński, W., Savarese, S., Zhou, Y., and Xiong, C. (2022). Long document summarization with top-down and bottom-up inference.

[Raffel et al., 2023] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. (2023). Exploring the limits of transfer learning with a unified text-to-text transformer.

[See et al., 2017] See, A., Liu, P. J., and Manning, C. D. (2017). Get to the point: Summarization with pointer-generator networks.

[Song et al., 2019] Song, K., Tan, X., Qin, T., Lu, J., and Liu, T.-Y. (2019). Mass: Masked sequence to sequence pre-training for language generation.

[Zhang et al., 2020] Zhang, J., Zhao, Y., Saleh, M., and Liu, P. J. (2020). Pegasus: Pre-training with extracted gap-sentences for abstractive summarization.

[Zhao et al., 2020] Zhao, Y., Saleh, M., and Liu, P. J. (2020). Seal: Segment-wise extractive-abstractive long-form text summarization.

[Zhou et al., 2018] Zhou, Q., Yang, N., Wei, F., Huang, S., Zhou, M., and Zhao, T. (2018). Neural document summarization by jointly learning to score and select sentences.