Ingegneria dei Dati 2025/2026

Homework 2

Silvia Mucci

https://github.com/silmucci/IDD_HW2.git

Indice

1 Introduzione

L'obiettivo dell'Homework 2 è la realizzazione di un programma capace di indicizzare e ricercare file di testo in formato `.txt` utilizzando la libreria `Apache Lucene`. Ogni documento è descritto da due campi principali:

- `title`: il nome del file;
- `content`: il contenuto testuale del file.

Il sistema permette di interrogare l'indice da console con una sintassi semplice e intuitiva del tipo:

> `title:<termine>` oppure `content:<termine>` o con una serie di termini

I risultati vengono restituiti in ordine di rilevanza, mostrando i documenti che contengono i termini cercati o le frasi esatte, ordinati in base al punteggio di similarità calcolato da Lucene.

2 Struttura e Funzionamento del Sistema

Il sistema è composto da due classi Java principali:

- `TXTIndexer.java`, che si occupa della creazione dell'indice e dell'indicizzazione dei file di testo;
- `TXTSearcher.java`, che gestisce l'interrogazione dell'indice e la visualizzazione dei risultati.

Durante la fase di indicizzazione, il programma legge i file presenti nella cartella `file_txt/`, crea per ciascuno un documento Lucene e assegna i due campi `title` e `content`. Entrambi i campi vengono salvati nell'indice, così da poter essere recuperati e visualizzati durante la ricerca.
La classe `TXTSearcher.java` apre successivamente l'indice creato e permette di eseguire ricerche da console. L'utente può specificare il campo su cui effettuare la ricerca che può essere `title` o `content` e digitare una o più parole chiave per avviare la ricerca. È inoltre possibile racchiudere i termini tra virgolette per eseguire una *phrase query*. Il sistema restituisce i documenti corrispondenti, ordinati in base al punteggio BM25 di Lucene, garantendo una risposta coerente con la query inserita. Questo lavoro dimostra come, attraverso un corretto utilizzo di Lucene, sia possibile costruire un motore di ricerca testuale efficiente e personalizzabile.

2.1 Fase di Indicizzazione e di Ricerca

La fase di indicizzazione costituisce il punto fondamentale del progetto che permette di trasformare un insieme di file testuali in una struttura dati efficiente e interrogabile.
Il programma avvia la procedura di indicizzazione, scorrendo tutti i file `.txt` presenti nella directory specificata. Tali file sono stati generati tramite uno script Python denominato `download_txt.py`, che scarica automaticamente i paper in formato `.pdf` dalla categoria *Information Retrieval* del portale `arXiv`. Successivamente, lo script estrae il contenuto testuale dai file PDF e lo salva in formato `.txt`, rendendolo pronto per la fase di elaborazione con Lucene.
Durante l'indicizzazione, ogni file viene processato e trasformato in un documento Lucene composto da due campi principali:

- `title` che contiene il nome del file e viene trattato come metadato;
- `content` che rappresenta l'intero testo del documento.

Entrambi i campi vengono memorizzati nell'indice, in modo da poter essere recuperati e visualizzati durante la fase di ricerca. L'indice finale viene salvato in una directory chiamata `index/`, che rappresenta la base su cui operano tutte le interrogazioni successive. Durante l'esecuzione, il programma mostra su console i file elaborati e, al termine, stampa il tempo totale di indicizzazione, calcolato come differenza tra l'istante iniziale e quello finale dell'operazione. In questo caso il file indicizzati risultano 35 e l'operazione risulta rapida ed efficiente.
Di seguito è riportato lo pseudocodice che descrive il funzionamento della classe `TXTIndexer`, responsabile della creazione dell'indice.

Listing 1: Pseudocodice per la creazione dell'indice (TXTIndexer)

```
Creazione indice (TXTIndexer)

Input:
  - directory_con_file_txt  percorso contenente i file .txt
  - directory_indice  percorso in cui creare lindice Lucene

1. Inizializza gli analyzer per i campi:
    - title  CustomAnalyzer
    - content  StandardAnalyzer
2. Crea un IndexWriter associato alla directory dellindice
3. Per ogni file f in directory_con_file_txt:
      a. Legge il contenuto del file
      b. Crea un nuovo documento Lucene
      c. Aggiunge campo "title" con il nome del file
      d. Aggiunge campo "content" con il testo del file
      e. Aggiunge il documento allindice
```

```
4. Stampa il numero di file indicizzati e il tempo totale di indicizzazione
```

Dopo la costruzione dell'indice, il sistema entra nella fase di ricerca, realizzata dalla classe `TXTSearcher.java`. Essa consente all'utente di inserire una query da console, indicando il campo su cui effettuare la ricerca (`title` o `content`), e di ottenere in risposta i documenti corrispondenti ordinati per rilevanza. Le query possono contenere termini singoli o frasi racchiuse tra virgolette per attivare la *phrase query*, ossia la ricerca di una sequenza esatta di parole. Il motore Lucene calcola per ogni documento un punteggio di rilevanza (*score*) BM25, un'evoluzione del classico schema TF–IDF, che consente di valutare in modo più accurato la rilevanza dei documenti rispetto alla query. Tale modello rappresenta la similarità di default adottata da Lucene.
I risultati vengono poi ordinati in base a tale punteggio e stampati su console, mostrando per ciascun documento:

- il titolo del paper;
- il nome del file `.txt`;
- il valore dello *score*;
- le statistiche generali della ricerca (numero di documenti trovati e tempo di risposta in millisecondi).

In caso di query non valide (ad esempio se non iniziano con `title:` o `content:`) o di formato errato, il sistema gestisce l'errore segnalando all'utente le opzioni corrette e interrompendo l'esecuzione. La classe `TXTSearcher` gestisce la fase di ricerca all'interno dell'indice. Il suo funzionamento è riassunto nel seguente pseudocodice.

Listing 2: Pseudocodice per la ricerca dei documenti (TXTSearcher)

```
Ricerca dei documenti (TXTSearcher)

Input:
  - directory_indice  percorso della directory dellindice Lucene
  - query_utente  query inserita da console

1. Apre lindice con un IndexReader
2. Inizializza un IndexSearcher per interrogare l'indice
3. Analizza la query_utente:
      - controlla che inizi con "title:" o "content:"
      - se il formato èerrato  mostra messaggio di errore
4. Crea un QueryParser per il campo specificato
5. Costruisce la Query (anche phrase query se racchiusa tra virgolette)
6. Esegue la ricerca con searcher.search(Query, MAX_RESULTS)
7. Recupera i risultati
```

```
8. Se nessun documento trovato:
       stampa "Nessun documento trovato"
   Altrimenti:
       per ogni risultato r:
          - estrae il documento
          - stampa titolo, file e valore di score
9. Stampa statistiche:
       - numero totale di documenti trovati
       - tempo di risposta in millisecondi
```

L'integrazione tra le due componenti, `TXTIndexer` e `TXTSearcher`, realizza quindi un sistema completo, stabile e scalabile per l'indicizzazione e la ricerca full-text.
La separazione tra fase di indicizzazione e fase di ricerca garantisce infine una gestione efficiente dei dati, consentendo di aggiornare o espandere l'indice senza compromettere le prestazioni complessive.

2.2 Analyzer utilizzati

Un aspetto fondamentale del progetto riguarda la scelta degli *analyzer*, strumenti che definiscono come il testo viene suddiviso in token e normalizzato durante le fasi di indicizzazione e ricerca. La selezione di un analyzer adeguato è determinante per la qualità dei risultati, poiché influenza direttamente il modo in cui Lucene interpreta e confronta le parole presenti nei documenti con quelle inserite nelle query.
Per il campo `title` è stato adottato un analyzer basato su `CustomAnalyzer`, costruito come segue:

```
Analyzer titleAnalyzer = CustomAnalyzer.builder()
            .withTokenizer(WhitespaceTokenizerFactory.class)
            .addTokenFilter(LowerCaseFilterFactory.class)
            .addTokenFilter(WordDelimiterGraphFilterFactory.class)
            .build();
```

Quindi, uesto analyzer è costruito tramite un `CustomAnalyzer` che utilizza come tokenizer la classe `WhitespaceTokenizerFactory`, la quale suddivide il testo in corrispondenza degli spazi bianchi. Successivamente, il filtro `LowerCaseFilterFactory` converte tutti i termini in minuscolo, garantendo una ricerca *case-insensitive*. Infine, il filtro `WordDelimiterGraphFilterFactory` gestisce parole composte o contenenti caratteri speciali, suddividendole in token più semplici (ad esempio `"AI-based"` diventa `"ai"`, `"based"`, `"ai-based"`). L'analyzer non applica alcun filtro di stopword, permettendo di mantenere tutti i termini, inclusi quelli più brevi o frequenti, caratteristica particolarmente utile per i titoli dei paper scientifici. In questo modo, la ricerca sui titoli risulta precisa e aderente al testo originale.
Per il campo `content`, invece, è stato impiegato uno `StandardAnalyzer`, che effettua la

tokenizzazione basata su regole Unicode e applica un filtro di stopword in lingua inglese. Questo tipo di analyzer è particolarmente adatto a testi lunghi e discorsivi, poiché elimina parole molto frequenti e poco significative (ad esempio “the”, “of”, “and”, “in”), migliorando la precisione e la qualità dei risultati restituiti. Grazie a questa configurazione, Lucene riesce a concentrarsi sui termini realmente informativi, riducendo il rumore semantico e aumentando la rilevanza dei documenti trovati.
La scelta di utilizzare analyzer differenti per i due campi consente di bilanciare efficacemente le esigenze del sistema: da un lato, una ricerca più flessibile e letterale sui titoli; dall’altro, un’analisi più semantica e raffinata sui contenuti testuali. Questa distinzione permette di ottenere risultati più coerenti con l’intento della ricerca, garantendo una combinazione equilibrata tra accuratezza e generalizzazione.

2.3 Query di test

Per valutare le prestazioni e la correttezza del sistema, è stata condotta una serie di test su query di diversa complessità, comprendenti sia termini singoli sia frasi esatte. Le ricerche sono state eseguite su entrambi i campi indicizzati (`title` e `content`), includendo anche combinazioni su più campi, al fine di analizzare la precisione, la copertura e i tempi di risposta del motore di ricerca.
I test hanno confermato il corretto funzionamento dell'indice e degli analyzer configurati. Le query applicate al campo `content` hanno restituito i documenti più pertinenti, mentre le ricerche sul campo `title` hanno garantito un riconoscimento preciso anche in presenza di variazioni tra maiuscole e minuscole. In particolare, le *phrase query* hanno mostrato un'elevata accuratezza nel recupero di espressioni esatte, dimostrando l'efficacia del sistema nel gestire sequenze di termini e relazioni contestuali tra le parole.
Durante ogni test, il programma ha stampato su console diverse informazioni utili per l'analisi dei risultati, tra cui:

- il numero totale di documenti trovati;
- i titoli dei paper corrispondenti e i relativi file `.txt`;
- il valore dello *score*, che rappresenta la rilevanza del documento rispetto alla query;
- il tempo di risposta:
- il numero complessivo di risultati.

Le immagini che seguono mostrano alcuni esempi di query eseguite, accompagnate da una breve descrizione dei risultati ottenuti.

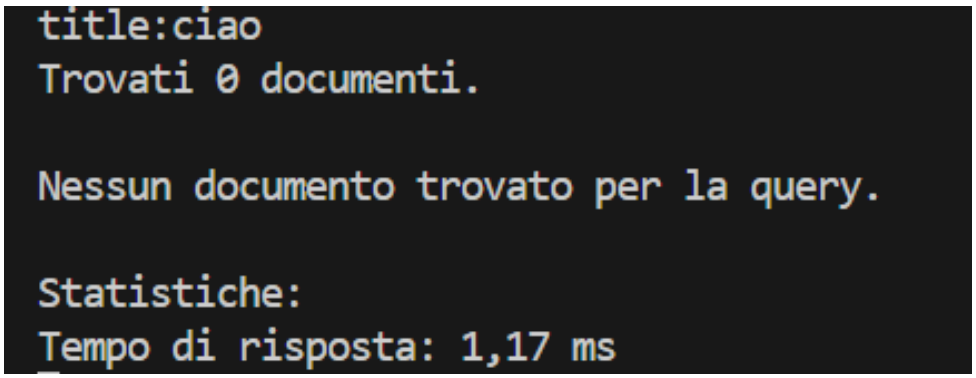

Figura 1: Esempio di query senza risultati: `title:ciao`.

Il sistema segnala correttamente l'assenza di documenti corrispondenti, poiché il titolo inserito non esiste. Viene mostrato in console il messaggio "Nessun documento trovato" e il tempo di risposta totale della ricerca.

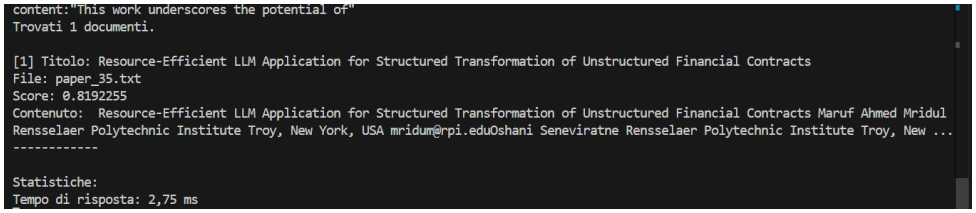

Figura 2: Esempio di phrase query sul contenuto: `content:"This work underscores the potential of"`.

In questo caso viene individuato un solo documento, con un punteggio di rilevanza elevato. La *phrase query* consente di recuperare espressioni esatte presenti nel testo, garantendo risultati di alta precisione.

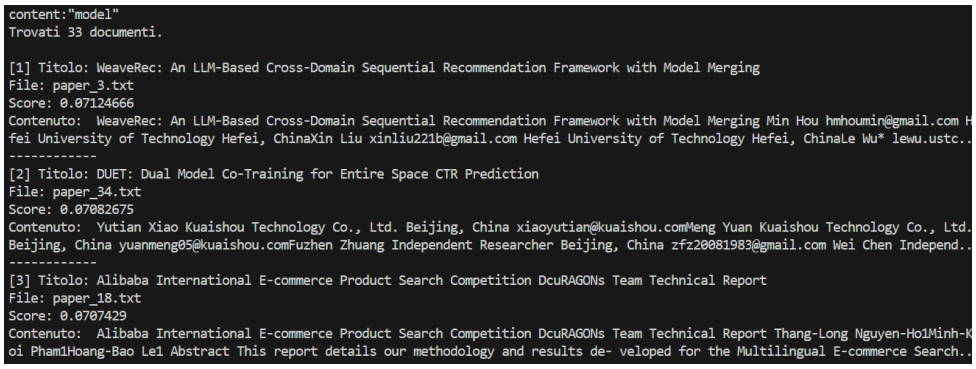

Figura 3: Ricerca per termine singolo nel contenuto: `content:model`.

La ricerca di un singolo termine restituisce 33 documenti pertinenti. Si osserva come il sistema ordini correttamente i risultati in base al punteggio di similarità calcolato secondo il modello BM25.

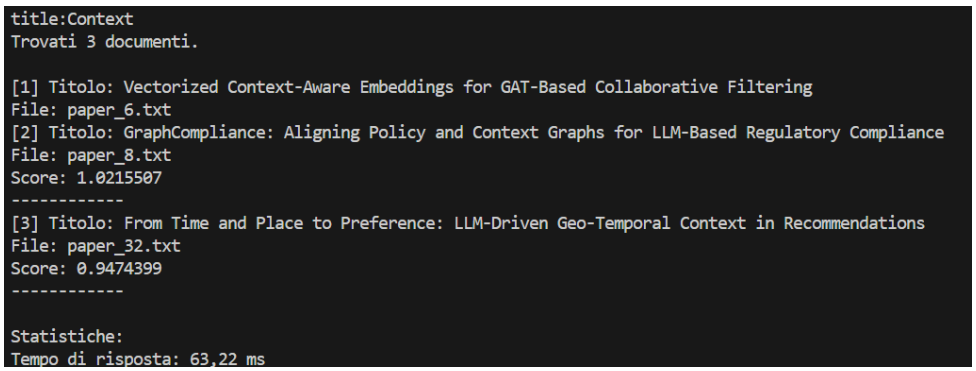

Figura 4: Esempio di query sul titolo: `title:Context`.

Vengono restituiti tre documenti pertinenti. La ricerca sul campo `title` si dimostra efficace anche in presenza di variazioni di maiuscole o minuscole.

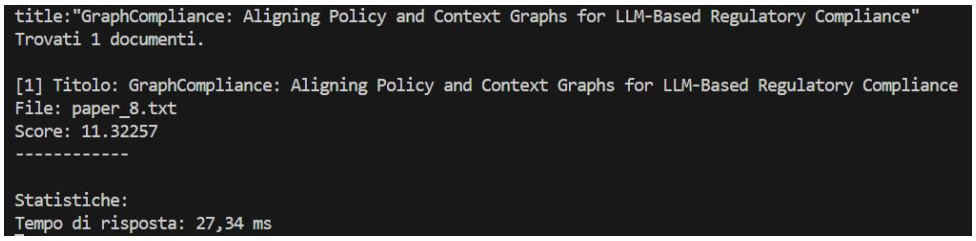

Figura 5: Esempio di phrase query completa sul titolo: `title:"GraphCompliance: Aligning Policy and Context Graphs"`.

Il sistema individua esattamente il documento corrispondente alla frase specificata. Questa prova dimostra la corretta gestione delle *phrase query* e l'allineamento tra tokenizzazione e struttura testuale dei titoli.

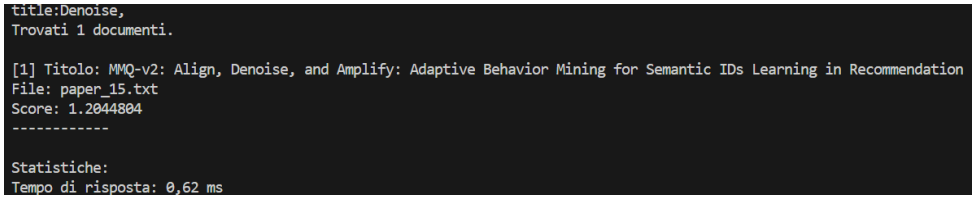

Figura 6: Ricerca per parola chiave nel titolo: `title:Denoise`.

La query restituisce un solo documento con tempo di risposta pari a 0.62 ms. Ciò conferma la rapidità del motore Lucene anche in presenza di indici di dimensioni non trascurabili.

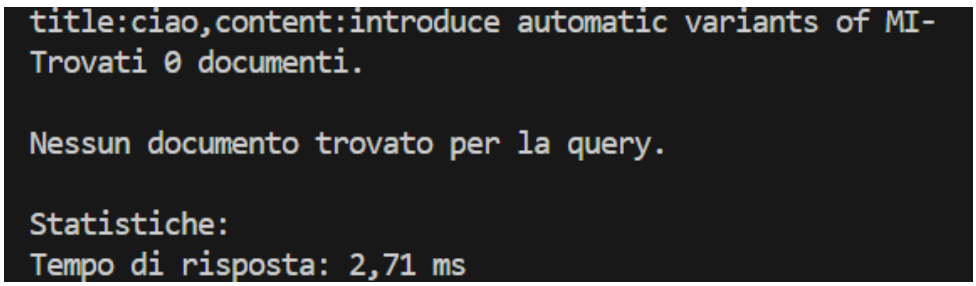

Figura 7: Esempio di query combinata su più campi: `title:ciao, content:introduce automatic variants of MI`.

In questa prova, il sistema combina più campi di ricerca ma non trova corrispondenze, poiché i termini specificati non compaiono nei documenti indicizzati. Il comportamento è corretto e coerente con la logica della query.

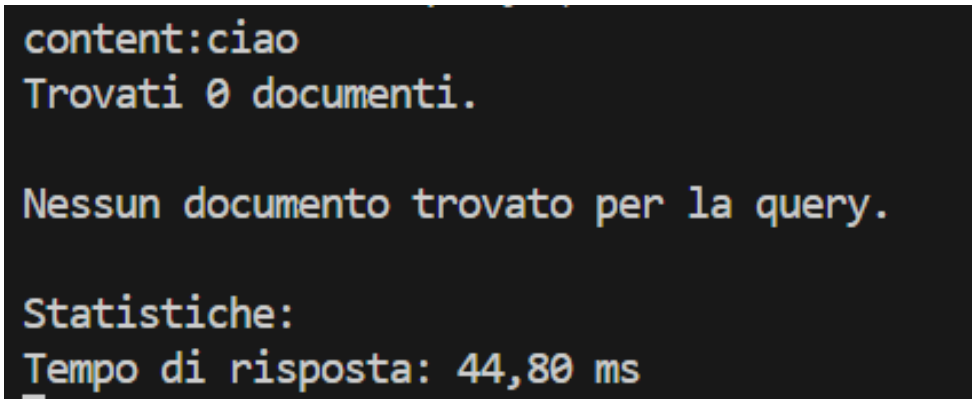

Figura 8: Query sul contenuto senza corrispondenze: `content:ciao`.

Il sistema mostra un messaggio di "nessun documento trovato" e mantiene un tempo di esecuzione inferiore ai 50 ms, evidenziando una gestione efficiente anche dei casi negativi.

Figura 9: Gestione di un campo non valido nella query (`tit:`).

Quando l'utente inserisce un campo non valido, il sistema intercetta l'errore e comunica i campi ammessi (`title` e `content`), garantendo robustezza e usabilità.

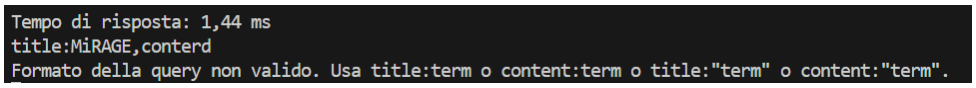

Figura 10: Esempio di errore di sintassi nella query: `title:MiRAGE, contered`.

In questo caso, la sintassi non rispetta il formato previsto (`campo:termine` o `campo:"frase"`). Il programma riconosce l'errore e fornisce un messaggio chiaro che guida l'utente nella corretta formulazione della query.
In tutti i casi, il sistema mostra un comportamento stabile e coerente, con un'ottima capacità di gestire query di varia complessità, fornendo risultati pertinenti e statistiche dettagliate sul tempo di esecuzione.
Le prove sperimentali hanno quindi dimostrato la robustezza e l'efficienza del motore di ricerca implementato, confermando la validità delle scelte progettuali adottate.

3 Conclusioni

Il progetto ha permesso di comprendere in modo concreto il funzionamento dei motori di ricerca testuali e i principi su cui si basa l'indicizzazione dei documenti. Grazie all'utilizzo della libreria `Apache Lucene`, è stato possibile realizzare un sistema in grado di indicizzare un insieme di file di testo e di consentirne la ricerca in modo rapido, preciso e affidabile. L'architettura sviluppata si è dimostrata semplice ma efficace, con una chiara separazione tra la fase di indicizzazione e quella di ricerca. Il sistema è inoltre facilmente estendibile: è possibile aggiungere nuovi campi (come autore, anno o parole chiave), integrare altri analyzer linguistici o collegarlo a un'interfaccia grafica per migliorare l'esperienza utente. In sintesi, il lavoro ha fornito una comprensione pratica di come costruire un motore di ricerca basato su Lucene, evidenziando l'importanza delle scelte progettuali e della gestione dei testi per ottenere risultati accurati e veloci.