UNIVERSITÀ DEGLI STUDI ROMA TRE

Ingegneria dei Dati

Homework 2 Individuale

Gaglione Giulia

Matricola 559057

Anno Accademico 2025/2026

Apache Lucene

Github URL: `https://github.com/giug2/idd-hw2-indiv-lucene.git`

L'obiettivo del secondo homework è sviluppare un sistema che permette di creare, indicizzare e successivamente ricercare documenti testuali utilizzando *Apache Lucene*.
Il sistema, dopo aver generato automaticamente un insieme di file di testo tramite Wikipedia, li indicizza e fornisce un'interfaccia web per consentire la ricerca testuale, valutandone le prestazioni in termini di Precision e Recall.

1.1 Tecnologie usate

Il progetto è stato sviluppato utilizzando tecnologie open-source la cui unione fornisce una pipeline completa per l'indicizzazione, la ricerca e la valutazione dei documenti testuali. Le principali componenti coinvolte sono descritte di seguito.

1.1.1 Apache Lucene

Apache Lucene è una libreria Java per la ricerca full-text e l'indicizzazione di grandi quantità di documenti testuali.
È il motore principale di gestione dell'indice: consente di analizzare i file di testo, tokenizzare i contenuti, rimuovere le stopwords e calcolare i punteggi di rilevanza durante la fase di ricerca.
Sono state sfruttate in particolare le seguenti classi e funzionalità:

- *IndexWriter* e *IndexWriterConfig* per la creazione dell'indice e la scrittura dei documenti;
- *DirectoryReader* e *IndexSearcher* per la ricerca;
- *QueryParser* e *TermQuery* per la definizione delle query;
- diversi *Analyzer* per gestire strategie di tokenizzazione differenti.

I diversi Analyzer utilizzati hanno permesso di sperimentare più configurazioni e confrontare la qualità dei risultati ottenuti.
La versione di Apache Lucene utilizzata per il progetto è la 10.3.1.

1.1.2 SpringBoot e Thymeleaf

La parte web dell'applicazione è stata sviluppata con *Spring Boot*, un framework Java che fornisce il supporto per la gestione dei controller.
Il livello di front-end è realizzato tramite *Thymeleaf*, un motore di *template HTML* che permette di generare dinamicamente le pagine a partire dai dati restituiti dal controller.
L'interfaccia web offre un form semplice per l'inserimento della query e una sezione per la visualizzazione dei risultati.

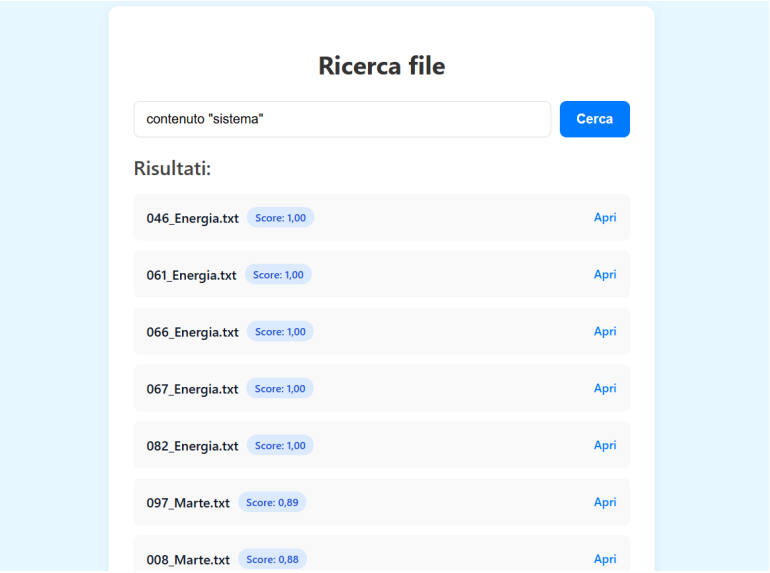


Figura 1.1: Aspetto della pagina web e della visualizzazione dei risultati.

1.1.3 Linguaggi e librerie

L'intero progetto è scritto in *Java 17*, per garantire compatibilità con le versioni recenti di Lucene e Spring.
Per la parte di generazione del dataset e l'automatizzazione dei test è stato utilizzato *Python 3*, insieme alla libreria wikipedia per la raccolta automatica dei testi.

1.1.4 Struttura del repository

Il repository è organizzato in questo modo:

```
create_txt.py             -> codice che genera il corpus dei documenti
test_query.py             -> codice che automaticizza i test
/docs/                    -> corpus dei documenti di testo generati
/index/                   -> directory contenente l'indice Lucene
/src/main/java/it/uniroma3/idd/
|-- AnalyzerFactory.java  -> classe che costumizza l'analyzer
|-- Indexer.java          -> indicizzazione dei file di testo
|-- LuceneWebApp.java     -> motore dell'applicazione
|-- Searcher.java         -> esecuzione delle query e calcolo metriche
|-- SearchController.java -> gestione della logica web e form di ricerca
|-- SearchResult.java     -> struttura dati per i risultati
\-- Stats.java            -> analisi statistica dell'indice
/resources/
|-- stopword.txt          -> contiene le stopwords della lingua italiana
\-/templates/             -> interfaccia web
  \-- index.html          -> pagina web
```

1.2 Generazione del dataset

La prima fase della pipeline consiste nella creazione automatica del dataset di documenti che costituiranno la base per l'indicizzazione in Lucene.
L'obiettivo è quello di creare un corpus di documenti realistico, costituito da testi di

lunghezza variabile, coerenti con i principali ambiti tematici forniti.
In questa fase viene eseguito uno script Python che utilizza la libreria *Wikipedia API* per scaricare automaticamente brevi estratti in lingua italiana relativi a una lista predefinita di argomenti (ad esempio Energia, Università, Computer).
Ogni estratto viene salvato in un file di testo all'interno della cartella *docs/* con nomenclatura del tipo "*001_ Energia.txt*"
Il contenuto del file ha una struttura semplice in cui il *nome*, definito nel titolo del file, è un argomento univoco facilmente riconoscibile, e il *contenuto* è il testo preso da Wikipedia, la cui lunghezza varia randomicamente in base al numero di frasi che vengono recuperate.
In totale vengono generati 200 file .txt, 10 per ciascuno dei 20 topic che vengono forniti.

1.3 Codice

1.3.1 LuceneWebApp

La classe *LuceneWebApp* costituisce il punto di avvio dell'applicazione e integra le due principali componenti del sistema: l'indicizzazione dei documenti e l'interfaccia di ricerca web.
È annotata con @SpringBootApplication, che abilita le funzionalità di Spring Boot.

1.3.2 AnalyzerFactory

Il codice definisce una classe *AnalyzerFactory* che fornisce un metodo statico per creare e restituire un oggetto Analyzer personalizzato utilizzando la libreria Apache Lucene.
Nel metodo *getCustomAnalyzer()*, viene costruito un CustomAnalyzer attraverso un pattern builder, specificando in sequenza le componenti di analisi linguistica. In particolare vengono definiti tre diversi Analyzer da poter testare:

1. Il primo analyzer utilizza il *WhitespaceTokenizerFactory*, che segmenta il testo in corrispondenza degli spazi bianchi, senza applicare regole linguistiche o di punteggiatura. Successivamente vengono applicati due filtri: *LowerCaseFilterFactory*,

che converte tutti i token in minuscolo, e *WordDelimiterGraphFilterFactory*, che gestisce la separazione o unione di parole composte, numeri e simboli.

2. Il secondo analyzer è costruito a partire dal tokenizer predefinito di Lucene (*standard*), che segmenta il testo in base alla grammatica generale della lingua, riconoscendo parole, numeri e simboli in modo flessibile. A questo vengono applicati diversi filtri: *lowercase*, che uniforma tutte le parole in minuscolo per evitare distinzioni tra maiuscole e minuscole; *stop*, che rimuove le parole comuni non significative (stopwords) specificate nel file stopword.txt; e *italianlightstem*, che riduce le parole alle loro radici lessicali attraverso uno stemming leggero ottimizzato per l'italiano.

3. Il terzo analyzer utilizza *ItalianAnalyzer*, un analyzer preconfigurato fornito direttamente da Apache Lucene e ottimizzato per la lingua italiana. Esso combina internamente un tokenizer standard con una serie di filtri linguistici, tra cui la conversione in minuscolo, la rimozione automatica delle stopword italiane integrate e l'applicazione dello stemming tramite l'algoritmo *Snowball* per l'italiano. Rispetto agli analyzer personalizzati, offre un compromesso ideale tra semplicità e prestazioni.

1.3.3 Indexer

La classe *Indexer* è responsabile della creazione dell'indice Lucene a partire dai documenti testuali presenti nella cartella *docs/*.
Nel metodo *createIndex()*, viene aperta una directory Lucene (FSDirectory) e configurato un IndexWriter in modalità CREATE, in modo da generare un nuovo indice da zero.
Il metodo *indexDocs()* attraversa ricorsivamente la cartella dei documenti tramite Files.walk(), filtrando i soli file di testo su cui richiama il metodo *indexDoc()*.
Quest'ultimo legge il contenuto del file e costruisce un oggetto Document Lucene composto da tre campi principali:

- *nome*: il nome del file indicizzato, salvato come *TextField* per poter essere ricercabile e memorizzato;

- *contenuto*: il testo completo del file, anch'esso analizzato e memorizzato;
- *argomento*: un campo derivato dal nome del file, estratto prendendo la parte successiva al primo underscore (_) e convertita in minuscolo, salvato come *StringField* (non analizzato ma memorizzato esattamente).

```
Indicizzato: 001_Acqua.txt (Argomento: acqua)
Indicizzato: 002_Supermercato.txt (Argomento: supermercato)
Indicizzato: 003_Nazione.txt (Argomento: nazione)
Indicizzato: 004_Java.txt (Argomento: java)
Indicizzato: 005_Web.txt (Argomento: web)
Indicizzato: 006_Java.txt (Argomento: java)
Indicizzato: 007_Java.txt (Argomento: java)
Indicizzato: 008_Marte.txt (Argomento: marte)
Indicizzato: 009_Supermercato.txt (Argomento: supermercato)
Indicizzato: 010_Python.txt (Argomento: python)
```

Figura 1.2: Elenco parziale dei file indicizzati.

Dopo aver costruito il documento, esso viene aggiunto all'indice tramite l'*IndexWriter*. Alla fine del processo, il programma stampa a console i file indicizzati, l'argomento estratto e il tempo totale impiegato per l'indicizzazione.

1.3.4 Searcher

La classe *Searcher* implementa la logica di ricerca e valutazione delle prestazioni all'interno dell'indice creato con Apache Lucene.
Essa utilizza lo stesso Analyzer definito dalla AnalyzerFactory, garantendo coerenza tra la fase di indicizzazione e quella di ricerca.
Per ciascun risultato, il metodo recupera il documento associato, ne estrae il nome e il punteggio di rilevanza, e verifica se l'argomento indicizzato corrisponde a quello atteso. Vengono quindi conteggiati i documenti rilevanti trovati e calcolate due metriche fondamentali della Information Retrieval Evaluation:

- Precision@10: proporzione di documenti rilevanti tra i primi 10 restituiti;
- Recall@10: proporzione di documenti rilevanti trovati rispetto a tutti quelli effettivamente rilevanti presenti nell'indice.

Infine, il programma stampa a console un riepilogo delle metriche e delle statistiche di ricerca, consentendo di valutare la qualità dei risultati ottenuti rispetto alla "ground truth" estratta.

1.3.5 SearcherController

La classe *SearchController* svolge il ruolo di ponte tra l'interfaccia web e la logica di ricerca Lucene, gestendo l'input utente, la validazione, l'esecuzione delle query e la presentazione dei risultati in modo modulare.

1.3.6 SearchResult

La classe *SearchResult* rappresenta una struttura dati semplice utilizzata per memorizzare e trasferire i risultati delle ricerche eseguite dal motore Lucene. Ogni oggetto di questa classe incapsula le informazioni essenziali di un documento restituito da una query, ovvero:

- *fileName*: il nome del file indicizzato corrispondente al documento trovato;
- *score*: il punteggio di rilevanza assegnato da Lucene al documento in base alla corrispondenza con la query.

1.3.7 Stats

La classe *Stats* ha la funzione di analizzare e fornire statistiche sull'indice Lucene precedentemente generato, offrendo una panoramica sul contenuto e sulla struttura dei dati indicizzati, stampando il numero totale di documenti indicizzati.

1.4 Query di test

Per verificare il corretto funzionamento dell'applicazione, sono stati sviluppati test automatizzati utilizzando uno script Python basato su *requests* e *BeautifulSoup*.
Lo script invia sequenzialmente un insieme di query all'applicazione Spring Boot e analizza i risultati restituiti.
I test sono stati eseguiti su tutti e tre i tipi di analyzer implementati, assicurando che

le differenze di tokenizzazione e indicizzazione fossero correttamente gestite.
Le query utilizzate nei test includono esempi come:

1. `contenuto "cibo"`: query sull'argomento Cibo;
2. `nome "cibo"`: query sull'argomento Cibo;
3. `contenuto "sistema"`: query sulla parola Sistema;
4. `nome "sistema"`: query sulla parola Sistema;
5. `contenuto "gocce"`: query sulla parola Gocce;
6. `nome "gocce"`: query sulla parola Gocce;
7. `contenuto ""`: query su stringa vuota;
8. `nome ""`: query su stringa vuota;
9. `contenuto "alto livello"`: query su stringa con due parole;
10. `nome "alto livello"`: query su stringa con due parole;
11. `nome "001"`: query su numero identificativo del file.

Queste query hanno permesso di valutare sia ricerche testuali complete che ricerche vuote o con caratteri speciali, garantendo una copertura ampia dei casi d'uso.

1.4.1 Test 1

```
public class AnalyzerFactory {
    public static Analyzer getCustomAnalyzer() throws IOException {
        return CustomAnalyzer.builder()
                .withTokenizer(WhitespaceTokenizerFactory.class)
                .addTokenFilter(LowerCaseFilterFactory.class)
                .addTokenFilter(WordDelimiterGraphFilterFactory.class)
                .build();
    }
}
```

Figura 1.3: Settaggio dell'analyzer utilizzato.

```
------- AVVIO STATISTICHE ------
Numero di documenti indicizzati: 200

Conteggio dei termini per ciascun campo:

- Campo: nome - Termini indicizzati: 221
- Campo: contenuto - Termini indicizzati: 1018
- Campo: argomento - Termini indicizzati: 20
--------------------------------
```

Figura 1.4: Statistiche del primo test.

- Tempo impiegato: 925 ms

1.4.2 Test 2

```
public class AnalyzerFactory {
    public static Analyzer getCustomAnalyzer() throws IOException {
        return CustomAnalyzer.builder()
                .withTokenizer("standard")
                .addTokenFilter("lowercase")
                .addTokenFilter("stop", "words", "stopword.txt")
                .addTokenFilter("italianlightstem")
                .build();
    }
}
```

Figura 1.5: Settaggio dell'analyzer utilizzato.

```
------- AVVIO STATISTICHE ------
Numero di documenti indicizzati: 200

Conteggio dei termini per ciascun campo:

- Campo: nome - Termini indicizzati: 200
- Campo: contenuto - Termini indicizzati: 933
- Campo: argomento - Termini indicizzati: 20
--------------------------------
```

Figura 1.6: Statistiche del secondo test.

- Tempo impiegato: 783 ms

1.4.3 Test 3

```java
public class AnalyzerFactory {
    public static Analyzer getCustomAnalyzer() throws IOException {
        return new ItalianAnalyzer();
    }
}
```

Figura 1.7: Settaggio dell'analyzer utilizzato.

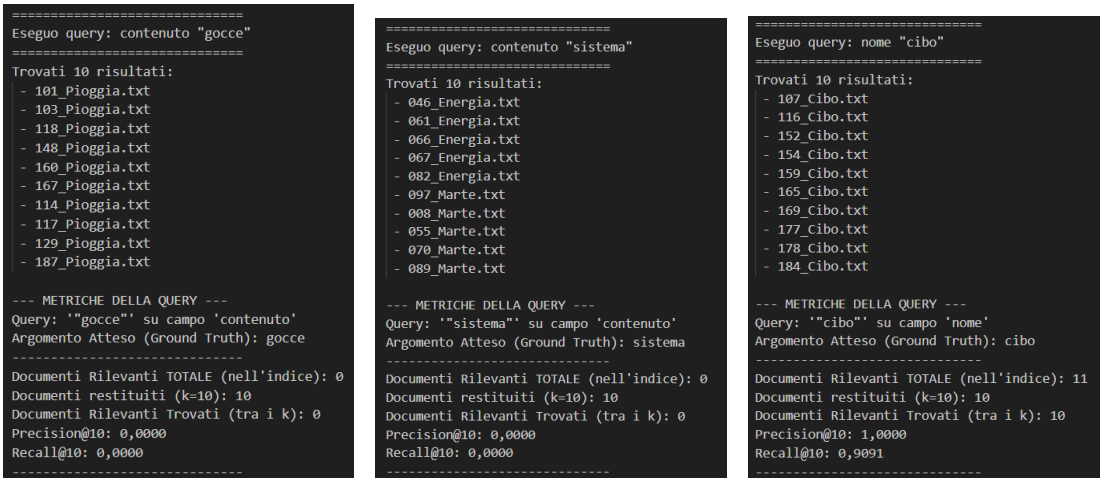
```
------- AVVIO STATISTICHE ------
Numero di documenti indicizzati: 200

Conteggio dei termini per ciascun campo:

- Campo: nome - Termini indicizzati: 200
- Campo: contenuto - Termini indicizzati: 871
- Campo: argomento - Termini indicizzati: 20
-------------------------------
```

Figura 1.8: Statistiche del terzo test.

- Tempo impiegato: 751 ms

1.5 Conclusioni

Dall'analisi dei test emerge che l'architettura di indicizzazione e ricerca implementata è pienamente funzionale. Il sistema infatti dimostra di essere in grado di recuperare con successo i documenti richiesti, operando correttamente sia su query basate sul campo nome (come nel test "cibo") sia sul campo contenuto (come nei test "gocce" e "sistema").



(a) Query "gocce" (b) Query "sistema" (c) Query "cibo"

Figura 1.9: Esempi dei risultati delle query di test.

L'analisi delle metriche di valutazione, tuttavia, richiede un'attenta interpretazione. I test relativi alle query sul contenuto ("gocce" e "sistema") mostrano valori di Precision@10 e Recall@10 pari a 0.0000. Questo risultato non indica un fallimento dell'indicizzazione, ma evidenzia un limite nella metodologia di valutazione adottata, data soprattutto dalla natura del dataset.

Infatti, per il calcolo della Precision e della Recall, sarebbe stato necessario disporre di un *ground truth* predefinito, come un insieme di etichette o tag pertinenti per ogni documento. Data la provenienza dei file, che sono stati generati da Wikipedia, tale etichettatura non era disponibile e una creazione manuale non era fattibile.

Si è quindi scelta una soluzione pragmatica: utilizzare come "Argomento Atteso" (Ground Truth) lo stesso termine di ricerca usato per creare i file, che corrisponde di fatto al loro nome (es. "cibo" per i file *_Cibo.txt).

Questa strategia funziona perfettamente per il test sul campo nome (query "cibo"), dove il ground truth e la query coincidono, portando a ottime metriche.

Al contrario, per le query sul contenuto (es. "gocce"), il sistema recupera correttamente file pertinenti (*_Pioggia.txt), ma il ground truth impostato è "gocce", non "Pioggia". Di conseguenza, i documenti recuperati, corretti per il contenuto, non corrispondono al ground truth, portando le metriche a zero.

In sintesi, i test confermano che l'indicizzazione e il recupero dei documenti funzionano

come atteso, mentre le metriche di Precision e Recall per le query sul contenuto risultano nulle unicamente a causa dell'assenza di un ground truth specifico per il contenuto. In prospettiva, l'utilizzo di un corpus di documenti più ricco e diversificato, possibilmente organizzato per domini tematici o categorie semantiche, accuratamente etichettate, avrebbe permesso di ottenere metriche di prestazione più significative e di evidenziare in modo più preciso i punti di forza e le eventuali aree di miglioramento dell'architettura di ricerca.