Indicizzazione e Ricerca di documenti HTML

Alessio Marinucci & Riccardo Felici

GitHub Repository

Indice

1 Introduzione

In questo homework l'obiettivo sarà quello di sviluppare un programma Java per l'indicizzazione di file '.html' disponibili presso l'indirizzo `all_htmls`. Si cercherà di creare indici per almeno due campi distinti. Oltre all'indicizzazione del campo abstract dei file, verranno presi in considerazione ulteriori campi quali il titolo e gli autori. Per ogni campo selezionato, sarà utilizzato un analyzer appropriato, al fine di garantire un'indicizzazione efficace e precisa.

Descrizione del Lavoro

Il programma Java dovrà essere in grado di leggere una query dalla console, interpellare l'indice generato e restituire i risultati pertinenti. La sintassi delle query sarà semplice e permetterà ricerche basate su specifici campi. Ad esempio, sarà possibile cercare per autore o abstract e formulare query di frase racchiuse tra virgolette per ricerche più precise.

Valutazione e Test

Il sistema sviluppato sarà sottoposto a una rigorosa fase di valutazione, utilizzando un set di dieci query diverse per analizzare sia l'efficacia dell'indicizzazione che la precisione delle risposte fornite. Ogni query sarà progettata per testare aspetti specifici del sistema, quali:

- `Copertura dei Campi Indicizzati`: Le query saranno formulate per verificare che tutti i campi indicizzati (es. titolo, contenuto, autore) vengano correttamente interrogati e restituiscano i risultati attesi.
- `Precisione e Rilevanza`: Sarà valutata la precisione del sistema, misurando la proporzione di documenti rilevanti restituiti rispetto al totale dei documenti trovati. Questo permetterà di capire se il sistema riesce a distinguere con efficacia i documenti rilevanti da quelli meno pertinenti.
- `Richiamo (Recall)`: Oltre alla precisione, verrà valutato il richiamo del sistema, ossia la capacità di individuare tutti i documenti rilevanti disponibili nell'indice. Una serie di query mirate aiuterà a determinare se l'indicizzazione è stata esaustiva e non ha trascurato documenti importanti.
- `Prestazioni e Tempi di Risposta`: Verranno monitorati i tempi di risposta per ciascuna query, per valutare l'efficienza del sistema. Sarà importante garantire che le ricerche siano eseguite in tempi ragionevoli, anche quando il set di dati cresce.
- `Gestione di Query Complesse`: Alcune query includeranno frasi racchiuse tra virgolette o combinazioni di termini per testare la capacità del sistema di gestire ricerche complesse e fornire risultati coerenti.
- `Robustezza e Gestione degli Errori`: Il sistema sarà anche testato per situazioni limite, come query vuote, termini inesistenti o caratteri speciali, per garantire che gestisca correttamente input anomali senza crash o errori significativi.

Al termine della fase di test, i risultati saranno raccolti e analizzati in dettaglio. Ogni query sarà accompagnata da una valutazione delle performance, un'analisi dei risultati ottenuti e delle osservazioni sulle possibili aree di miglioramento.

2 Tecnologie Utilizzate

- **Java**
  - *Descrizione*: Java è un linguaggio di programmazione orientato agli oggetti, ampiamente utilizzato per applicazioni di tipo enterprise, web, desktop e mobile.
  - *Ruolo nel Progetto*: Il progetto è scritto interamente in Java, fornendo la base per l'implementazione della logica di indicizzazione e ricerca.
- **Apache Lucene**
  - *Descrizione*: Apache Lucene è una libreria open-source per il recupero di informazioni (IR) e la ricerca full-text.
  - *Ruolo nel Progetto*: Viene utilizzato per creare un indice dei file HTML e per eseguire query di ricerca, consentendo una ricerca rapida ed efficiente.
- **Jsoup**
  - *Descrizione*: Jsoup è una libreria Java per il parsing di HTML, che consente di estrarre e manipolare dati da documenti HTML.
  - *Ruolo nel Progetto*: Viene utilizzata per leggere e analizzare i file HTML, estraendo contenuti come il titolo, l'autore e il testo del documento.
- **Maven**
  - *Descrizione*: Maven è uno strumento di gestione del progetto e di automazione della build per Java.
  - *Ruolo nel Progetto*: Facilita la gestione delle dipendenze e automatizza la compilazione del progetto.
- **Struttura di Directory Standard**
  - *Descrizione*: La struttura di directory standard facilita l'organizzazione del codice sorgente e delle risorse.
  - *Ruolo nel Progetto*: Organizza il codice sorgente e i file HTML in cartelle specifiche per una migliore manutenzione.
- **StandardAnalyzer (di Apache Lucene)**
  - *Descrizione*: È un analizzatore che suddivide il testo in termini utilizzando regole standard e rimuove le parole comuni ("stop words").
  - *Ruolo nel Progetto*: Viene utilizzato per analizzare il campo `abstract` durante l'indicizzazione e la ricerca.

- **CustomAnalyzer (di Apache Lucene)**
  - *Descrizione*: È un analizzatore personalizzato che consente di definire regole specifiche per la tokenizzazione e il filtraggio dei termini, combinando diversi componenti (come tokenizzatori e filtri) per creare una pipeline di analisi su misura. Questo consente una maggiore flessibilità e adattabilità per applicazioni che richiedono personalizzazioni linguistiche, filtri specifici o ottimizzazioni per un dominio di riferimento.
  - *Ruolo nel Progetto*: Viene utilizzato per analizzare i campi `title` e `author` durante l'indicizzazione e la ricerca, offrendo una pipeline di analisi configurata per ottenere risultati più accurati in base alle esigenze del dominio applicativo. Con il `CustomAnalyzer`, è possibile configurare tokenizzazione, rimozione di *stop words*, normalizzazione (ad es. conversione in minuscolo) e applicare tecniche come lo stemming o la lemmatizzazione.
- **BufferedReader**
  - *Descrizione*: Una classe Java per leggere il testo da un input di caratteri, bufferizzando i caratteri per fornire un'efficienza maggiore.
  - *Ruolo nel Progetto*: Utilizzato per leggere le query inserite dall'utente attraverso la console. Piuttosto che caricare l'intero contenuto di un file HTML in memoria, il BufferedReader può leggere il file riga per riga, ottimizzando l'uso della memoria e migliorando le prestazioni.

3 Pipeline del Progetto

Per lo svolgimento del progetto è stato necessario seguire una serie di passaggi, qui sotto descritti: questa documentazione descrive una pipeline sistematica e ben organizzata per la realizzazione di un sistema di indicizzazione e ricerca di file HTML, come si vede in Figura 1.

3.1 Fasi della Pipeline

1. **Acquisizione dei Dati**
   (a) **Input dei File:** Raccogliere i file HTML dalla directory designata (`all_htmls`).
   (b) **Verifica dei File:** Controllare che i file siano accessibili e formattati correttamente.
2. **Parsing dei Dati**
   (a) **Lettura dei File HTML:** Utilizzare una libreria come Jsoup per leggere il contenuto dei file HTML.
   (b) **Estrazione dei Dati:** Estrarre informazioni pertinenti come:
   - Titolo del documento.
   - Autore o altri metadati.
   - Contenuto principale (ignorando i tag HTML).
   - Abstract del documento.
3. **Indicizzazione**
   (a) **Preparazione dei Dati:** Pulire e formattare il contenuto estratto per l'indicizzazione.
   (b) **Utilizzo di un Analizzatore:** Applicare un analizzatore (ad esempio, `StandardAnalyzer` di Apache Lucene) per suddividere il testo in termini, rimuovendo parole comuni (*stop words*).
   (c) **Creazione dell'Indice:** Utilizzare Apache Lucene per creare un indice strutturato che memorizzi i dati indicizzati per un accesso rapido.
4. **Querying**
   (a) **Input dell'Utente:** Ricevere query dall'utente tramite la console.
   (b) **Interpretazione della Query:** Analizzare la sintassi della query per determinarne il significato e i parametri.
   (c) **Esecuzione della Query:** Interrogare l'indice creato per cercare risultati pertinenti basati sulla query fornita.

5. **Restituzione dei Risultati**

   (a) **Formattazione dei Risultati:** Presentare i risultati della ricerca in un formato leggibile e comprensibile, includendo i metadati rilevanti (ad esempio, titolo, autore, estratti del contenuto).

   (b) **Visualizzazione:** Mostrare i risultati nella console, fornendo opzioni per ulteriori ricerche o filtri.

6. **Valutazione e Test**

   (a) **Testing delle Query:** Eseguire una serie di test con query predefinite per valutare l'efficacia dell'indicizzazione e la pertinenza dei risultati.

   (b) **Analisi delle Prestazioni:** Monitorare i tempi di indicizzazione e di risposta alle query per identificare potenziali aree di miglioramento, utilizzando anche differenti analizzatori per identificare il migliore tra essi.

Figure 1: Pipeline dell'architettura utilizzata

4 Struttura del codice

Per svolgere il progetto è stata necessaria la creazione di due file .java : **HTMLIndexer.java** e **HTMLSearcher.java**.

4.1 Pseudocodice per l'Indicizzazione HTML

Di seguito è riportato lo pseudocodice per l'indicizzazione dei file HTML:

Pseudocodice per il file HTMLIndexer.java

```
classe HTMLIndexer {
    scrittore: IndexWriter
    metodo HTMLIndexer(percorsoIndice) {
        apriDirectory(percorsoIndice)
        creaAnalyzerStandard()
        configuraIndexWriter()
    }
    metodo indexHtmlFiles(percorsoHtml) {
        directoryHtml = creaFile(percorsoHtml)
        fileHtmls = listaFileHtml(directoryHtml)
        se fileHtmls è non nullo allora
            per ogni fileHtml in fileHtmls:
                chiama indexDocument(fileHtml)
            fine per
        fine se
    }
    metodo indexDocument(fileHtml) {
        htmlDoc = analizzaFileHtml(fileHtml)
        documento = nuovo Documento()
        titolo = estraiTitolo(htmlDoc)
        autore = estraiAutore(htmlDoc)
        contenuto = estraiContenuto(htmlDoc)
        aggiungiCampo(path,title,author,content,abstract,relevant)
    }
    metodo findAuthor(){
        trova la lista di autori, cercando nei nodi dell'HTML
        }
    metodo findAbstract(){
        trova i vari abstract, cercando nei nodi dell'HTML
    }
    metodo close() {
        chiudiScrittore(scrittore)
    }
}
```

Il programma `HTMLIndexer` si occupa di indicizzare i file HTML in un indice Lucene. Ecco una spiegazione passo dopo passo delle operazioni che esegue:

1. **Inizializzazione dell'Indexer:**
   - Crea un oggetto `IndexWriter` per gestire l'indicizzazione dei documenti.
   - L'indice viene creato in una directory specificata da `indexPath`.
   - Viene utilizzato un `Analyzer` predefinito di tipo `StandardAnalyzer` per l'analisi dei testi.
   - La configurazione dell'`IndexWriter` viene impostata in modalità `CREATE`, che sovrascrive eventuali indici preesistenti, ogni volta che si esegue il programma.
2. **Indicizzazione dei file HTML:**
   - Il metodo `indexHtmlFiles` prende in input un percorso `allHtmls` che punta alla directory contenente i file HTML.
   - Vengono verificati i file HTML presenti nella directory, filtrando solo quelli con estensione `.html`.
   - Per ogni file HTML trovato, viene chiamato il metodo `indexDocument`, che crea un documento Lucene per l'indicizzazione.
3. **Creazione del documento Lucene:**
   - Per ogni file HTML, il metodo `indexDocument` effettua il parsing del documento HTML utilizzando la libreria `Jsoup`.
   - Vengono estratti i seguenti campi dal documento HTML:
     - `title` (titolo del documento)
     - `author` (autore del documento, estratto tramite il metodo `findAuthor`)
     - `content` (contenuto del documento, estratto come testo)
     - `abstract` (abstract del documento, estratto tramite il metodo `findAbstract`)
   - I campi estratti vengono aggiunti come `Field` al documento Lucene, con i rispettivi tipi di analisi (`TextField` per il contenuto testuale e `StringField` per il percorso del file).
   - Il documento viene quindi aggiunto all'indice tramite `IndexWriter`.
4. **Metodi ausiliari:**
   - Il metodo `findAuthor` cerca l'autore nel documento HTML utilizzando selettori CSS. Se non viene trovato un autore, restituisce "Unknown".
   - Il metodo `truncateIfNeeded` tronca una stringa se la sua lunghezza supera i 32000 caratteri (limite massimo per i campi Lucene).
   - Il metodo `findAbstract` cerca l'abstract nel documento HTML, prima in un meta tag e poi all'interno di vari tag nel corpo del documento. Se non trova un abstract, restituisce "No abstract available".

5. **Chiusura dell'Indexer:**

   - Una volta completata l'indicizzazione, il metodo `close` chiude l'oggetto `IndexWriter` e stampa un messaggio di conferma.

6. **Esecuzione del programma:**

   - Il metodo `main` avvia il processo di indicizzazione.
   - Specifica il percorso dell'indice e della directory contenente i file HTML.
   - Una volta completata l'indicizzazione, viene chiamato `close` per chiudere l'Indexer.

4.2 Pseudocodice per la ricerca nell'indice

Nel paragrafo seguente è riportato lo pseudocodice per la ricerca nell'indice:

Pseudocodice per il file HTMLSearcher.java

```
classe HTMLSearcher {
    searcher: IndexSearcher
    titleParser, abstractParser, authorParser: QueryParser
    totalDocs: long

    metodo HTMLSearcher(percorsoIndice) {
        apriDirectory(percorsoIndice)
        creaIndexReader()
        creaIndexSearcher()
        totalDocs = leggiNumeroDocumenti()
        titleParser = creaQueryParser("title",CustAnalyzer())
        abstractParser = creaQueryParser("abstract",StandAnalyzer())
        authorParser = creaQueryParser("author",CustAnalyzer())
    }
    metodo search(queryStr) {
        iniziaMisurazioneTempo()
        booleanQueryBuilder = creaQueryBoolean()

        per ogni termine in dividiQuery(queryStr) {
            campo, campoQuery = dividiTermine(termine)
            query = creaQuery(campo, campoQuery)
            aggiungiQueryA(booleanQueryBuilder, query)
        }
        risultati = eseguiRicerca(searcher,
        booleanQueryBuilder.build(), 10)
        misuraTempoFine()
        stampaRisultati(risultati)
        calcolaPrecisioneRichiamo(risultati)
    }
    metodo close() {
        chiudiIndexReader(searcher)
    }
}
```

Il codice Java che segue è un'implementazione di un sistema di ricerca basato su Lucene. Ecco una spiegazione dettagliata di cosa fa:

1. **Inizializzazione della classe HTMLSearcher**:
   - Quando la classe `HTMLSearcher` viene inizializzata, viene creato un `IndexSearcher` che serve a effettuare la ricerca all'interno dell'indice.
   - Viene anche aperto un `IndexReader` per leggere i documenti salvati nell'indice, e si recupera il numero totale di documenti (`totalDocs`).
   - Vengono creati tre analizzatori personalizzati per i campi `title`, `abstract`, e `author`, utilizzando diversi `Analyzer`.
2. **QueryParser per ciascun campo**:
   - `titleParser`: un `QueryParser` per il campo "title", che usa un analizzatore personalizzato (custom) che separa le parole e le rende minuscole.
   - `abstractParser`: un `QueryParser` per il campo "abstract", che usa uno `StandardAnalyzer`.
   - `authorParser`: un `QueryParser` per il campo "author", che usa un altro analizzatore personalizzato (custom) simile a quello per il "title".
3. **Metodo `search(queryStr)`**:
   - Viene inizializzato un timer che calcola il tempo di esecuzione della ricerca.
   - La funzione `search` riceve una query (stringa) da parte dell'utente e crea un oggetto `BooleanQuery` che serve a costruire la query composta.
   - La query è divisa in base ai vari termini separati da virgola. Ogni termine è un campo (`title`, `abstract`, `author`) seguito da un valore.
   - Viene analizzato ogni campo specificato nella query, e viene creato un oggetto `Query` per ciascun campo utilizzando il rispettivo `QueryParser`.
   - Le query per ogni campo possono essere combinate tra loro per formare una query booleana (`BooleanQuery`), in cui le condizioni devono essere tutte soddisfatte (`BooleanClause.Occur.MUST`).
4. **Esecuzione della ricerca**:
   - La query booleana viene eseguita usando `IndexSearcher.search` e viene limitata a 10 risultati.
   - Viene misurato il tempo di esecuzione della ricerca, registrato in nanosecondi.
5. **Visualizzazione dei risultati**:
   - I risultati della ricerca sono stampati sullo schermo, inclusi i documenti trovati (con `path`, `title`, `author`, `abstract`, e `content`).
   - Per ogni documento trovato, se il campo `relevant` è `true`, viene conteggiato come documento rilevante.
   - Viene inoltre stampato un punteggio con un campo `Score` che, in base a quanto è stata precisa la query, è piu o meno alto.

6. **Calcolo di precisione e richiamo**:
    - La `precisione` è calcolata come il numero di documenti rilevanti trovati diviso il numero di documenti restituiti dalla ricerca.
    - Il `richiamo` è calcolato come il numero di documenti rilevanti trovati diviso il numero totale di documenti nell'indice.
7. **Statistiche e copertura dei campi**:
    - Dopo la ricerca, vengono stampati i seguenti dati:
        - Tempo di risposta della ricerca (in millisecondi).
        - Precisione e richiamo.
        - Indica se i vari campi (`title`, `abstract`, `author`) sono stati interrogati.
8. **Metodo** `close()`:
    - Il metodo `close` chiude l'`IndexReader` per liberare le risorse.
9. **Metodo** `main`:
    - Il metodo `main` permette all'utente di inserire query da riga di comando.
    - L'utente inserisce una query come stringa, la quale viene poi passata al metodo `search` per eseguire la ricerca.
    - Il ciclo continua fino a quando l'utente non digita "exit".

4.3 Tipi di Analyzer utilizzati

In questa sezione, vengono approfonditi i tipi di `Analyzer` utilizzati nel progetto e le ragioni che hanno motivato la scelta di ciascuno. Gli `Analyzer` in Apache Lucene svolgono un ruolo fondamentale nel processo di indicizzazione e ricerca, poiché gestiscono la tokenizzazione e la normalizzazione del testo, influenzando direttamente l'accuratezza e la pertinenza dei risultati di ricerca.

4.3.1 Custom Analyzer per il campo `title`

Per il campo `title` è stato utilizzato un `CustomAnalyzer` composto da un `WhitespaceTokenizer` e da due filtri: `LowerCaseFilter` e `WordDelimiterGraphFilter`. Questo approccio risponde all'esigenza di gestire titoli che possono contenere parole composte, caratteri speciali o diverse convenzioni di maiuscolo/minuscolo. Di seguito, vengono illustrate le caratteristiche di ciascun componente:

- **WhitespaceTokenizer**: Questo tokenizer suddivide il testo in token separati da spazi bianchi. Poiché i titoli spesso contengono parole composte o singoli termini significativi, è vantaggioso preservare i termini senza frammentare ulteriormente il testo, come farebbe uno `StandardTokenizer`.
- **LowerCaseFilter**: Questo filtro converte tutti i termini in minuscolo, rendendo il campo `title` case-insensitive, una scelta utile per evitare problemi di matching causati da variazioni nella capitalizzazione.
- **WordDelimiterGraphFilter**: Questo filtro suddivide i termini composti e riconosce termini uniti da delimitatori comuni, come trattini o punti. Questa configurazione aumenta la capacità di ricerca in caso di variazioni nella scrittura, favorendo un matching più accurato.

L'uso di questo `CustomAnalyzer` nel campo `title` migliora l'adattabilità ai diversi formati dei titoli, garantendo una ricerca più flessibile e accurata, anche per titoli complessi o con delimitatori.

4.3.2 Analyzer Standard per il campo `abstract`

Per il campo `abstract`, che contiene testo discorsivo, è stato scelto lo `StandardAnalyzer`. Questa scelta si basa sulla capacità dell'analyzer di gestire testi complessi in linguaggio naturale.

- **StandardAnalyzer**: Questo analyzer utilizza filtri per normalizzare il testo e rimuovere le stopwords (parole comuni come "e", "il", "di", ecc.) che non sono indicative per una ricerca. Questa configurazione è particolarmente efficace nel campo `abstract`, dove occorre estrarre informazioni utili da frasi complete e ignorare parole comuni che ridurrebbero la precisione della ricerca.

L'uso dello `StandardAnalyzer` permette di ottenere una segmentazione accurata delle informazioni più rilevanti, evitando il rumore causato dalle stopwords e dalla variazione nella capitalizzazione.

4.3.3 Custom Analyzer per il campo `author`

Per il campo `author` è stato configurato un altro `CustomAnalyzer`, simile a quello usato per il titolo ma senza il filtro `WordDelimiterGraphFilter`. Questo approccio è stato scelto per mantenere la ricerca case-insensitive, preservando al contempo l'integrità dei nomi composti.

- **WhitespaceTokenizer**: Anche in questo caso, il tokenizer è stato scelto per mantenere intatti i nomi e cognomi degli autori, compresi i nomi composti (es., "Van Gogh").
- **LowerCaseFilter**: Questo filtro rende la ricerca case-insensitive, permettendo di cercare gli autori senza preoccuparsi della capitalizzazione.

Questa configurazione assicura una ricerca accurata e tollerante rispetto a variazioni di maiuscole e minuscole, mantenendo l'integrità dei nomi propri.
La scelta di diversi `Analyzer` per i campi `title`, `abstract` e `author` risponde all'esigenza di ottimizzare la ricerca in modo specifico per ciascun campo. Il `CustomAnalyzer` per `title` e `author` garantisce la ricerca indipendentemente da maiuscole/minuscole e delimitatori, mentre lo `StandardAnalyzer` per `abstract` è ottimizzato per il linguaggio naturale. Ogni `Analyzer` è stato selezionato per rispondere alle caratteristiche del campo applicato, migliorando così l'efficacia complessiva del sistema di ricerca.

5 Possibili Scenari di Utilizzo

In questa sezione vengono analizzati alcuni possibili scenari di utilizzo del nostro indice appena creato:

5.1 Ricerca Rapida e Mirata nei Documenti HTML

L'indice permette di effettuare ricerche rapide e mirate all'interno di un grande numero di documenti HTML. Senza un indice, la ricerca richiederebbe una scansione completa di ogni singolo documento, risultando lenta e inefficiente. Con l'indice, invece, la ricerca di termini specifici (come titoli, autori o parole chiave) diventa immediata, migliorando notevolmente l'efficienza. Questo è particolarmente utile per chi gestisce grandi collezioni di contenuti HTML, come articoli di notizie, blog o documenti di ricerca.

5.2 Applicazioni Web o Mobile per una Navigazione Facile

Nei contesti web o mobile, l'indice diventa uno strumento fondamentale per migliorare l'esperienza utente. Per esempio, un'applicazione che offre contenuti in formato HTML, come un e-book, un sito di notizie, o una piattaforma di articoli accademici, può usare l'indice per fornire una funzionalità di ricerca istantanea, che consente all'utente di trovare rapidamente quello che sta cercando, senza rallentamenti o interruzioni nell'interazione con il contenuto.

5.3 Analisi dei Dati e Reportistica

L'indice non è utile solo per la ricerca, ma anche per l'analisi dei dati contenuti nei documenti. Permette di estrarre informazioni specifiche in modo automatizzato, come la frequenza di occorrenza di certe parole o la distribuzione di contenuti tra vari autori o categorie. Questo scenario è utile per chi desidera analizzare grandi quantità di testo e generare report su tendenze o informazioni specifiche, per esempio, in contesti di ricerca accademica o marketing.

6 Analisi dei Risultati

Di seguito vengono analizzati i risultati ottenuti dopo una fase di indicizzazione e una fase di ricerca dell'indice che è stato creato utilizzando Apache Lucene. I risultati riflettono la capacità del sistema di recupero di informazioni di rispondere efficacemente alle query proposte, con l'obiettivo di valutare la qualità delle risposte rispetto ai documenti indicizzati e alle metriche di valutazione stabilite.

6.1 Fase di Indicizzazione

La fase di indicizzazione ha comportato la preparazione dei documenti e la creazione di un indice strutturato che possa essere rapidamente e accuratamente interrogato. Durante questa fase, ogni documento è stato analizzato con specifici *analyzer* e trasformato in una rappresentazione indicizzata con l'obiettivo di preservare il contenuto rilevante per la ricerca. Alcuni documenti hanno incluso un campo specifico, chiamato `relevant`, il cui scopo è di indicare la rilevanza del documento in base a criteri predeterminati (es., contenuto, autore, ecc.). Tuttavia, è stata riscontrata l'assenza di alcuni valori di rilevanza nei documenti indicizzati, il che può aver influito sull'analisi di precisione e richiamo nella fase di ricerca.

6.2 Fase di Ricerca

Durante la fase di ricerca, il sistema è stato testato con una serie di query progettate per interrogare diversi campi, tra cui il titolo, l'abstract e l'autore. La struttura delle query ha previsto una combinazione di criteri tramite un'operazione di *BooleanQuery*, permettendo così di specificare condizioni su uno o più campi.

6.3 Metriche di Valutazione

Per la valutazione della qualità dei risultati, sono state utilizzate le seguenti metriche:

- **Precisione**: rappresenta la frazione di documenti rilevanti rispetto a tutti i documenti recuperati per una determinata query. In altre parole, misura la qualità dei risultati mostrati al termine della ricerca.
- **Richiamo**: rappresenta la frazione di documenti rilevanti trovati rispetto al totale dei documenti rilevanti presenti nell'indice. Questa metrica è indicativa della capacità del sistema di trovare effettivamente tutti i documenti che soddisfano la query.

6.4 Analisi dei Risultati di Precisione e Richiamo

In base ai risultati della ricerca, è stato osservato che i valori di precisione e richiamo risultano bassi, indicativi di una possibile presenza in numero abbastanza piccolo di documenti rilevanti alla configurazione dell'indice.

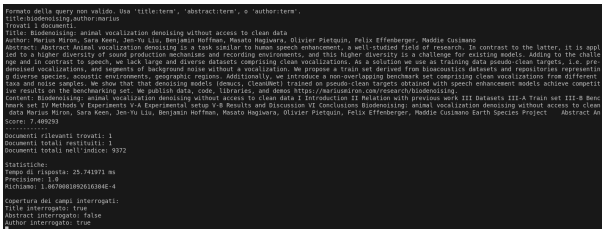
6.5 Copertura dei Campi Interrogati

Un'analisi della copertura dei campi interrogati ha permesso di verificare l'efficacia del sistema nel rispondere a query su campi specifici:

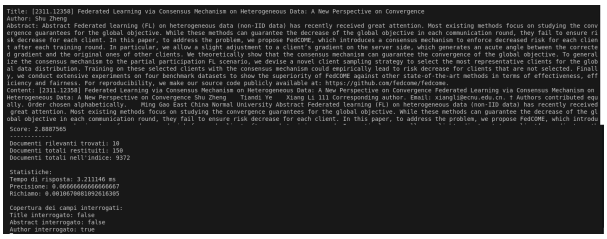
- **Titolo**: la ricerca ha dimostrato un buon recupero delle informazioni quando il titolo è stato specificato, confermando l'efficacia dell'analyzer personalizzato utilizzato per questo campo.
- **Abstract**: il recupero dei risultati in base all'abstract ha evidenziato alcune differenze a livello di accuratezza della ricerca, probabilmente a causa di una diversa configurazione dell'analyzer (StandardAnalyzer) rispetto a quello utilizzato per il titolo.
- **Autore**: l'interrogazione per autore ha prodotto risultati coerenti con l'indicizzazione, mantenendo un'efficacia adeguata durante il processo.

6.6 Esempi di query

In questa sezione vengono mostrati graficamente alcuni esempi di query svolte sul sistema.



(a) Esempio di query combinata `title:biodenoising,author:marius`

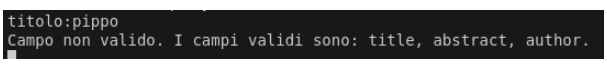


(b) Esempio di query author `author:zheng`

(c) Esempio di query abstract `abstract:dataset`

(d) Esempio di query senza documenti `title:app,abstract:radio,author:john`

(e) Esempio di query title `title:community`



(f) Esempio di query non valida `titolo:pippo`

Figure 2: Esempi di query

7 Conclusioni

Di seguito vengono sintetizzati i risultati ottenuti durante il processo di indicizzazione e ricerca attraverso Apache Lucene, e vengono discussi gli insegnamenti tratti dall'esperienza di utilizzo del sistema. Il sistema sviluppato ha permesso di esplorare a fondo le potenzialità del motore di ricerca Apache Lucene, nonché le sfide che si affrontano nel costruire un sistema di recupero delle informazioni efficiente e preciso.

7.1 Sommario dei Risultati

Il sistema di ricerca realizzato ha permesso di gestire un vasto indice di documenti, utilizzando campi specifici come `title`, `author` e `abstract` per affinare le query di ricerca. Durante il processo di ricerca, sono stati ottenuti risultati significativi che hanno evidenziato come l'ottimizzazione della query possa influenzare la precisione e il richiamo dei documenti. Le statistiche calcolate, come la precisione e il richiamo, sono state cruciali per valutare l'efficacia del sistema, sebbene alcuni limiti siano emersi, in particolare riguardo alla gestione della pertinenza dei documenti.

7.2 Nozioni Apprese

Di seguito sono riportate le principali nozioni apprese durante lo sviluppo del sistema di ricerca:

- **Ottimizzazione delle Query**: La costruzione di query ben definite e ottimizzate è essenziale per ottenere risultati di ricerca pertinenti. L'uso di operatori logici e la separazione dei termini in base ai vari campi (titolo, autore, abstract) si è rivelato fondamentale per ottenere risultati di qualità.
- **Gestione della Pertinenza dei Documenti**: La definizione dei documenti pertinenti è stata una sfida importante. Le tecniche di valutazione come la precisione e il richiamo sono stati strumenti utili, ma è necessario un processo accurato di annotazione dei dati per definire correttamente i documenti pertinenti.
- **La Potenza di Apache Lucene**: Apache Lucene è stato un motore di ricerca molto potente, grazie alla sua capacità di gestire grandi volumi di dati e alla sua efficienza nell'indicizzazione e nella ricerca. L'analizzatore personalizzato e l'uso di filtri di tokenizzazione hanno migliorato la qualità delle ricerche.
- **Bilanciamento tra Precisione e Richiamo**: La gestione del trade-off tra precisione e richiamo è stata una delle sfide principali. Ottimizzare uno dei due parametri potrebbe compromettere l'altro, quindi è stato necessario bilanciare entrambi per ottenere risultati significativi.

7.3 Sviluppi Futuri

In futuro, il sistema di ricerca potrebbe essere migliorato su vari aspetti:

- Introduzione di un terzo metodo Java dedicato alla valutazione continua della qualità delle ricerche. Questo metodo potrebbe avere una duplice funzione: monitorare le query effettuate dagli utenti e tracciare le loro preferenze e abitudini di ricerca. Un'analisi statistica delle query più frequenti, insieme ai risultati più cliccati o visualizzati, consentirebbe al sistema di acquisire una comprensione più approfondita delle necessità dell'utenza, permettendo di personalizzare i risultati per riflettere tali preferenze. Questa personalizzazione potrebbe, ad esempio, modificare il ranking dei documenti per migliorare la rilevanza rispetto ai trend di ricerca degli utenti.

- Integrazione di un meccanismo di *auto-apprendimento* per ottimizzare il sistema in modo dinamico e adattativo. Questo sistema, basato su tecniche di *machine learning*, permetterebbe al motore di ricerca di apprendere dai risultati meno soddisfacenti per gli utenti. Ogni volta che una query restituisce un numero ridotto di risultati, o risposte giudicate poco rilevanti, il sistema potrebbe registrare tali eventi come indicatori di bassa qualità e analizzarli. In questo modo, l'algoritmo potrebbe identificare automaticamente errori di indicizzazione, come documenti con metadati inaccurati o contenuti mal classificati, e risolvere questi problemi attraverso un riadattamento dei parametri di indicizzazione. Questa capacità di auto-apprendimento potrebbe essere ulteriormente potenziata implementando un ciclo di aggiornamento continuo dei dati di indicizzazione e delle regole di query. Il sistema potrebbe utilizzare algoritmi di apprendimento supervisionato o non supervisionato per rilevare pattern in query precedenti e prevedere i miglioramenti necessari. Ad esempio, per query che frequentemente non restituiscono risultati rilevanti, il sistema potrebbe ampliare automaticamente i sinonimi dei termini di ricerca o perfezionare le regole di stemming, così da garantire una maggiore copertura dei concetti ricercati. Di conseguenza, il motore di ricerca diventerebbe progressivamente più accurato, in grado di adattarsi ai cambiamenti nelle preferenze dell'utenza e di rispondere con maggiore precisione alle nuove richieste.

- Creazione di un modulo di feedback diretto, attraverso cui gli utenti segnalano l'efficacia dei risultati o suggeriscono miglioramenti. Integrando i dati di feedback con il meccanismo di auto-apprendimento, il sistema potrebbe affinare ulteriormente le risposte e migliorare costantemente l'esperienza di ricerca.

- Definizione di un programma Java che supporti l'operazione di indicizzazione estendibile a un dominio di file differenti dal formato HTML.