Università degli Studi di Roma Tre

Ingegneria dei Dati

Homework 2

Creazione di un Motore di Ricerca tramite un Indice Lucene

Antonio Lanza, Daniel Luca

Project repository: https://github.com/AntonioSouls/homework2-IDD

1. Introduzione

Nell'ambito della Data Science, è molto frequente dover lavorare con grosse moli di dati, estrapolando informazioni specifiche da questi.
Motivo per cui, spesso, risulta di necessaria importanza avere degli strumenti che consentano una ricerca ed un accesso veloce ai dati che si hanno a disposizione.
Questo rappresenta la motivazione principale per cui nasce l'homework propostoci e per cui si è manifestata l'utilità di tale relazione.
L'obiettivo di tale homework, in primo luogo, era quello di realizzare un programma Java che indicizzasse una grande quantità di file HTML sulla base di almeno un paio di campi. In secondo luogo, ci poneva la sfida di creare una console che permettesse all'utente di inserire una query e, a seguito di tale inserimento, interrogasse gli indici precedentemente creati per poi stampare il risultato.

Dunque, come si può ben intuire da quanto scritto, tale assegnazione aveva l'obiettivo di porci di fronte a sfide reali nel mondo della Data Science, permettendoci così, di affrontare problematiche che ci spronassero a trovare la miglior soluzione al problema sopra elencato.
In questo breve elaborato, quindi, si andrà a presentare il nostro lavoro riguardante questo secondo homework del corso "Ingegneria dei Dati", descrivendo dettagliatamente le soluzioni da noi ideate e le motivazioni che ci hanno spinto a preferire tali soluzioni.

2. Strumenti utilizzati

L'inizio di tale progetto è stato caratterizzato dalla necessità di scaricare in locale circa 10K file HTML da un folder predisposto, in modo da poter favorire l'indicizzazione. La raccolta di tali file consisteva in numerosi articoli a tema scientifico scaricati da ar5iv.org e utilizzati da tutti gli studenti del corso nel precedente homework.

Tramite l'IDE di sviluppo IntelliJ IDEA, è stato poi realizzato il progetto Java per la creazione degli indici e la gestione delle query. La scelta di tale IDE è dovuta al fatto che IntelliJ rappresenta l'IDE migliore sul mercato per sviluppare progetti Java essendo dotato di un'interfaccia easy-to-use che facilita la gestione di tutti i package del progetto e l'interazione con GitHub.

In particolare, per svolgere le attività di indicizzazione e ricerca, si è usufruito dell'API open-source Apache Lucene, ovvero uno strumento di information retrieval molto rapido ed efficiente, spesso utilizzato per la realizzazione di motori di ricerca. Grazie all'utilizzo di questa API e dei metodi da essa offerti, è stato possibile realizzare un'indicizzazione precisa dei documenti scaricati

3. Svolgimento

La prima principale attività di creazione degli indici è stata interanente gestita da una classe denominata ***HTMLIndexer***.
Il principale metodo di questa classe è ***indexDocs*** in cui vengono effettivamente creati gli indici sui documenti.
Il metodo ***indexDocs*** segue i seguenti passi:

- istanziamento di un analyzer personalizzato per analizzare i campi dei documenti
- creazione di un writer, opportunamemte configurato con l'analyzer personalizzato, per poter scrivere i documenti nell'indice
- parsing dei file HTML ed estrazione dei campi title, content, abstract, e authors tramite la libreria Jsoup
- creazione di un documento Lucene per ogni file HTML e aggiunta dei campi estratti al documento
- aggiunta all'indice del documento appena creato

La classe ***HTMLIndexer*** contiene anche un metodo main che richiama indexDocs passandogli come parametro la directory in cui si intende salvare gli indici.

Un ruolo fondamentale è stato assunto dagli analyzer che effettuano operazioni di tokenizzazione e filtraggio del testo.
Apache Lucene è dotato già di una serie di analyzer pronti all'uso per elaborare il testo in svariate modalità; fornisce però anche la posssibilità di costruire i propri analyzer personalizzati, combinando opportunamente quelli predefinti o creando token e filtri personalizzati.
Avendo scelto di indicizzare i file HTML considerando quattro differenti campi, è risultato fondamentale applicare le giuste elaborazioni linguistiche e tokenizzazioni specifiche per ogni campo in modo da ottimizzare l'indicizzazione e garantire ricerche accurate ed efficienti.
Ciò è stato realizzato tramite una classe apposita, denominata ***PersonalAnalyzer***.
Tale classe è composta da quattro metodi, ognuno dei quali svolge il ruolo di analyzer per il proprio campo:

- *getTitleAnalyzer*:

```
public static Analyzer getTitleAnalyzer() {
    try {
        return CustomAnalyzer.builder()
            .withTokenizer(WhitespaceTokenizerFactory.class)
            .addTokenFilter(LowerCaseFilterFactory.class)
            .addTokenFilter(WordDelimiterGraphFilterFactory.class)
            .addTokenFilter(EnglishPossessiveFilterFactory.class)
            .addTokenFilter(PorterStemFilterFactory.class)
            .build();
    }
    catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

- *getAuthorsAnalyzer*:

```
public static Analyzer getAuthorsAnalyzer() {
    try {
        return CustomAnalyzer.builder()
            .withTokenizer(WhitespaceTokenizerFactory.class)
            .addCharFilter(PatternReplaceCharFilterFactory.class,"pattern"
            ",", "replacement", " ")
            .addTokenFilter(LowerCaseFilterFactory.class)
            .build();
    }
    catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

- *getContentAnalyzer*:

```
public static Analyzer getContentAnalyzer() {
    try {
        return CustomAnalyzer.builder()
            .withTokenizer(StandardTokenizerFactory.class)
            .addCharFilter(HTMLStripCharFilterFactory.class)
            .addTokenFilter(LowerCaseFilterFactory.class)
            .addTokenFilter(WordDelimiterGraphFilterFactory.class)
            .addTokenFilter(EnglishPossessiveFilterFactory.class)
            .addTokenFilter(PorterStemFilterFactory.class)
            .build();
    }
    catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

- *getAbstractAnalyzer*:

```
public static Analyzer getAbstractAnalyzer() {
    try {
        return CustomAnalyzer.builder()
            .withTokenizer(StandardTokenizerFactory.class)
            .addCharFilter(HTMLStripCharFilterFactory.class)
            .addTokenFilter(LowerCaseFilterFactory.class)
            .addTokenFilter(WordDelimiterGraphFilterFactory.class)
            .addTokenFilter(EnglishPossessiveFilterFactory.class)
            .addTokenFilter(PorterStemFilterFactory.class)
            .build();
    }
    catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

I quattro metodi appena mostrati utilizzano il ***CustomAnalyzer*** builder per costruire l'analyzer personalizzato, partendo da token e filtri predefiniti.
In alternativa, sarebbe stato possibile realizzare gli stessi analyzer estendendo la classe astratta ***Analyzer*** ed effettuando l'override del metodo astratto ***createComponents***.
Si può notare come, negli analyzer di titolo e autori, è impiegato il ***WhitespaceTokenizer*** per tokenizzare il testo basandosi quindi esclusivamente sugli spazi bianchi (o tab). In tali campi, infatti, risulta fondamentale preservare la punteggiatura e alcuni caratteri speciali.
Diversamente nei campi contenuto e abstract, dove la punteggiatura ed i caratteri speciali non sono significativi per la ricerca, viene rimossa la punteggiatura e vengono gestiti i caratteri speciali, separandoli da token. Per questa ragione, per tali due campi, si è usufruito dello ***StandardTokenizer***.
Per tutti i campi si è impiegato il ***LowerCaseFilter***, convertendo tutti i caratteri in minuscolo in modo da rendere le ricerche case-insensitive.
Altri filtri utilizzati sono il ***WordDelimiterGraphFilter*** che gestisce alcuni delimitatori e camel case, il ***PorterStemFilter*** che apllica lo stemming di alcune parole riducendole alla radice e l'***EnglishPossessiveFilter*** grazie al quale vengono rimossi i possessivi inglesi.
Content e abstract contengono inoltre l'***HTMLStripCharFilter*** per la rimozione dei tag HTML dal testo.

L'attività di ricerca dei documenti all'interno degli indici, creati da ***HTMLIndexer***, è stata realizzata da una classe ***Main*** che si comporta come un vero e proprio motore di ricerca.
All'interno di essa infatti vengono eseguite le seguenti funzionalità:

- gestione dell'interazione con l'utente
- esecuzione delle query all'indice
- stampa dei risultati

4. Conclusioni

In conclusione, tale trattazione ha evidenziato più volte come l'obiettivo del progetto fosse quello di realizzare un motore di ricerca per i documenti del folder predisposto. Superfluo sottolineare che si richiedeva sia un'indicizzazione che una ricerca celeri ed efficienti.
Nella realizzazione di tale progetto, dunque, si è insistito molto sulla sua efficienza andando ad ottenere buoni risultati.
Per quanto riguarda la **creazione dell'indice**, infatti, il tempo di esecuzione è risultato essere poco più di *5 minuti*, tempo che abbiamo reputato molto buono soprattutto dopo essersi confrontati anche con i risultati ottenuti da altri colleghi.
La fase di **esecuzione delle query** e ricerca dei risultati, invece, ha dei tempi che si aggirano su una media di *1 sec per query*, risultato appena soddisfacente (dovuto forse alla complessità delle query eseguite), ma sicuramente migliorabile in quanto si ritiene che tale tempistica si dovrebbe aggirare nell'ordine dei millisecondi.

Ottimi risultati sono stati rinvenuti anche per quanto riguarda la correttezza delle risposte alle interrogazioni.
In questo frangente, si è proceduto seguendo due vie:

1. Creazione di una classe di test in cui sono state testate **TermQueries** e **PhraseQueries**;
2. Calcolo della precision delle queries effettuate da console;

Per il primo punto, si è deciso di testare la ricerca per ogni campo. Dunque, si è preso come riferimento un documento specifico, intitolato "Biodenoising:animal vocalization denoising without access to clean data" che è stato sfruttato come parametro di ricerca. Nello specifico, sono stati effettuati:

- UN TEST SUL CAMPO TITLE = In cui è stata testata la TermQuery "Biodenoising", verificando che la ricerca restituisse quello specifico documento che conteneva quello specifico termine. Su questo campo, inoltre, è stata testata anche la PhraseQuery "animal vocalization denoising" con la quale si è verificato che venisse restituito il suddetto documento cercando un'intera frase sul campo "title" invece che una singola parola. Entrambi i test hanno restituito il documento richiesto, dimostrando la correttezza della ricerca sul campo "title";
- UN TEST SUL CAMPO CONTENT = In cui è stata testata la TermQuery "denoIsing" e la PhraseQuery "pairing Vocalization of noise". In entrambi i casi, è stata effettuato una ricerca esclusivamente sul contenuto ed è stato restituito, come risultato di tali query, il suddetto documento atteso, dimostrando come la ricerca avvenisse correttamente anche per il campo "content". Inoltre, con questi suddetti test, si è dimostrato anche il corretto funzionamento degli Analyzer dimostrando il corretto parsing della query e la corretta analisi della stessa al fine di fornire un buon risultato per la ricerca effettuata;
- UN TEST SUL CAMPO AUTHORS = In questo caso, si è testata la TermQuery "Hoffman" e la PhraseQuery "Benjamin Hoffman" andando a ricercare esclusivamente sul campo "authors". L'obiettivo di tale test era quello di verificare che la ricerca sull'indice restituisse, con entrambe le query, il suddetto documento e ciò è avvenuto. Questo fatto ha sottolineato ancora come gli Analyzer funzionassero correttamente e come la ricerca avvenisse senza errori;
- UN TEST SUL CAMPO ABSTRACT = Anche in quest'ultimo caso è stato effettuato sia un test sulla TermQuery "denoising" che uno sulla PhraseQuery "Vocalization denoising". Questo test, in linea con tutti gli altri effettuati, ci ha restituito vari documenti tra cui quello sopra citato, andando a confermare la correttezza già stimata con i precedenti test;

La fase di verifica della correttezza dei risultati restituiti dal sistema, però, come già specificato, non si è limitata soltanto a questa classe di test. Abbiamo, infatti, anche testato le varie query da console.

In particolar modo, tra le varie query effettuate, hanno avuto particolare rilevanza le seguenti:

- "Benjamin Hoffman" effettuata solo sul campo "authors". I risultati di questa interrogazione sono stati i titoli di due documenti che, andando a verificare manualmente, sono risultati essere due documenti il cui autore è effettivamente Hoffman;
- "Benjamin Hoffman" effettuata su tutti i campi. I risultati di questa interrogazione sono stati 5 documenti che, effettuando una ricerca manuale, non sono risultati essere solamente i due documenti sopra citati, ma anche documenti in cui Hoffman veniva citato;

Come già sottolineato, oltre a queste query, ne sono state effettuate altre che hanno verificato la correttezza dei documenti restituiti. Per ogni query effettuata, infatti, sono stati analizzati manualmente i documenti restituiti verificando che, in almeno uno dei campi su cui veniva effettuata la ricerca, fossero contenuti i termini rilevanti della PhraseQuery effettuata sulla console.
Dopo una serie di tentativi, è risultato difficile creare un test che verificasse correttamente che, per ogni singola query, venissero restituiti tutti i documenti rilevanti.

In conclusione, però, è possibile affermare che, con una buona stima, l'indicizzazione è avvenuta correttamente ed i documenti restituiti dall'indice a seguito di una ricerca sono tutti pertinenti alla specifica query effettuata.