



Relazione Finale Big Data Schema Matching

2020-2021

Emanuele Spezia

Giuseppe Brescia



Introduzione

All'interno del progetto NOAH, realizzato in collaborazione con i dottorandi di Roma Tre (*Roger Voyat, Valerio Cetorelli, Luca Emili*), abbiamo avuto modo di risolvere problemi attinenti alle tematiche Big Data, sperimentandone le metodologie e analizzandone le tecnologie. Il progetto di ricerca NOAH, mira allo sviluppo di un sistema per la creazione semiautomatica di pipeline di elaborazione dati web end-to-end. Le pipeline estraggono e integrano continuamente informazioni da più siti sfruttando la ridondanza dei dati pubblicati sul Web. Le informazioni contenute al loro interno possono essere estratte e memorizzate all'interno di un database NoSQL, così da poter essere utilizzate per effettuare interessanti analisi. Siamo stati, così, inseriti all'interno del progetto con il compito di implementare la fase 2 delle 3 totali che andranno a comporre l'intera pipeline.

Obiettivi

Dopo l'estrazione dei dati provenienti dalla fase 1 della pipeline, essi necessitano di essere normalizzati prima di poter essere integrati. Il nostro primo task all'interno del progetto, è stato quindi, quello di normalizzare i dati estratti da diversi siti e memorizzarli all'interno del DBMS NoSQL ClickHouse. Successivamente, dopo la fase di normalizzazione, siamo passati alla fase di interpretazione dei dati, dove il secondo task è stato quello di riconoscere la semantica dei dati assegnandovi la giusta label semantica. Una volta interpretati i dati, vi si possono effettuare delle analisi. L'ultimo task, infine, è stato quello di cercare il matching tra dati estratti da siti diversi ma di stesso dominio. Gli obiettivi possono dunque essere sintetizzati come segue:

1. Effettuare queries al DBMS NoSQL Clickhouse
2. Memorizzazione ed estrazione dati in forma vettoriale
3. Ricerca relazioni fra vettori di dati estratti da siti web utilizzando un algoritmo di matching
4. Test risultato finale

Tecnologie usate

I. Database - ClickHouse



ClickHouse è un database management system (DBMS) open-source di tipo column-oriented utile per online analytical processing (OLAP). Sviluppato dall'azienda russa Yandex, consente l'analisi di dati che vengono aggiornati in tempo reale ed è progettato per alte prestazioni. Supporta la linear scalability, la fault tolerance ed è dotato di funzionalità e meccanismi di sicurezza a livello aziendale.¹ Inoltre, risulta essere molto semplice configurare e da utilizzare in quanto permette l'utilizzo di query su dati real-time attraverso linguaggio SQL. Non necessita di file di configurazione complessi e solitamente la distribuzione e la replica dei dati non causa problemi. La documentazione risulta essere molto dettagliata e vanta una numerosa community.

II. Ambiente di sviluppo - IntelliJ IDEA CE



L'ambiente di sviluppo che abbiamo utilizzato è IntelliJ IDEA, sviluppato da JetBrains. IntelliJ è un IDE per il linguaggio di programmazione Java disponibile anche in licenza Apache. Ogni aspetto di IntelliJ IDEA è stato progettato per massimizzare la produttività degli sviluppatori. L'assistenza alla codifica intelligente e il design ergonomico ne rendono lo sviluppo non solo produttivo ma anche divertente. Dopo che IntelliJ IDEA ha indicizzato il tuo codice sorgente, offre un'esperienza straordinariamente veloce e intelligente fornendo suggerimenti pertinenti in ogni contesto: completamento del codice istantaneo e intelligente, analisi del codice al volo e strumenti di refactoring affidabili. Strumenti mission-critical come sistemi di controllo della versione integrati e un'ampia varietà di linguaggi e framework supportati sono tutti a portata di mano, senza problemi di plug-in inclusi.²

III. Linguaggio di programmazione - Java 17



Java è un linguaggio di programmazione e una piattaforma di elaborazione sviluppati da Sun Microsystems nel 1995. Esiste un numero notevole di applicazioni e siti Web, in aumento ogni giorno, che funzionano esclusivamente se è stato installato Java. Java è veloce, sicuro e affidabile. Dai portatili ai datacenter, dalle console per videogiochi ai computer altamente scientifici, ai telefoni cellulari e a Internet, Java è onnipresente.

IV. Structured Query Language



SQL (acronimo di Structured Query Language) è il linguaggio di interrogazione più diffuso tra quelli usati per l'interazione con i principali Database Management Systems (DBMS). E' diventato uno standard per i DBMS relazionali ma viene largamente utilizzato da molti altri DBMS, tra cui ClickHouse che ne supporta un linguaggio simile.

Specifiche Tecniche

Per l'implementazione del progetto e lo sviluppo del codice, nonché test e debugging, sono stati utilizzati due calcolatori portatili, le cui specifiche sono elencate qui di seguito:

Specifiche Hardware Portatile 1:

OS: macOS Big Sur 11.5

CPU: Intel® Core™ i7 dual-core 3.3 GHz

RAM: 16 GB 2133 MHz LPDDR3

GPU: Intel Iris Graphics 550 1536 MB

Specifiche Hardware Portatile 2:

OS: Pop!_OS 21.04

CPU: Intel® Core™ i5-4210U 1.70GHz

RAM: 8 GB DDR3

GPU: GeForce GT 820M

Implementazione progetto

Data import

I dati per la realizzazione del progetto sono stati resi reperibili su ClickHouse. Per accedere al server remoto sul quale risiede ClickHouse e quindi poter effettuare delle query, bisogna effettuare un tunnel, instaurando una connessione SSH, tramite il comando da terminale:

```
ssh -L 8123:speedup.dia.uniroma3.it:8123 -N  
bigdata2021_2@speedup.dia.uniroma3.it
```

Una volta instaurata la connessione con il server remoto è possibile chiamare i servizi REST di ClickHouse per effettuare delle query ad esempio digitando da terminale il comando:

```
curl -i -X POST -d "SELECT * FROM  
extracted_data.c5e54033df0a4d17801808da46506300" http://localhost:8123
```

In alternativa è possibile visualizzare i dati tramite la GUI di ClickHouse, la quale può essere acceduta inserendo nella barra degli indirizzi del browser il comando:

http://localhost:8123/play

http://localhost:8123

SELECT * FROM extracted_data.c5e54033df0a4d17801808da46506300

Run (Ctrl+Enter) ✓

timestamp	page_id	page_url
2021-06-24 18:55:59	0082	file:/local/cetorelli/exp_naruto/SWDE/movie/movie_hollywood(2000)/0082.htm
2021-06-24 18:56:24	0095	file:/local/cetorelli/exp_naruto/SWDE/movie/movie_hollywood(2000)/0095.htm
2021-06-24 18:56:04	0105	file:/local/cetorelli/exp_naruto/SWDE/movie/movie_hollywood(2000)/0105.htm
2021-06-24 18:56:17	0107	file:/local/cetorelli/exp_naruto/SWDE/movie/movie_hollywood(2000)/0107.htm
2021-06-24 18:56:01	0122	file:/local/cetorelli/exp_naruto/SWDE/movie/movie_hollywood(2000)/0122.htm
2021-06-24 18:56:07	0123	file:/local/cetorelli/exp_naruto/SWDE/movie/movie_hollywood(2000)/0123.htm
2021-06-24 18:56:31	0124	file:/local/cetorelli/exp_naruto/SWDE/movie/movie_hollywood(2000)/0124.htm
2021-06-24 18:56:20	0127	file:/local/cetorelli/exp_naruto/SWDE/movie/movie_hollywood(2000)/0127.htm
2021-06-24 18:56:21	0135	file:/local/cetorelli/exp_naruto/SWDE/movie/movie_hollywood(2000)/0135.htm
2021-06-24 18:56:13	0141	file:/local/cetorelli/exp_naruto/SWDE/movie/movie_hollywood(2000)/0141.htm
2021-06-24 18:55:59	0154	file:/local/cetorelli/exp_naruto/SWDE/movie/movie_hollywood(2000)/0154.htm
2021-06-24 18:56:07	0157	file:/local/cetorelli/exp_naruto/SWDE/movie/movie_hollywood(2000)/0157.htm

Da notare che oltre al nome del DB verso cui effettuiamo la query (extracted_data), vi è anche la notazione della specifica tabella alla quale accediamo (ID dopo il punto, c5e5403...). All'interno del DB, la maggior parte delle colonne presentava una label non significativa, identificata da una sequenza di caratteri che non ne specificava il contenuto (es. iill, iilr, iirl, etc.). Inoltre, le tabelle presenti nel database erano composte da molte colonne, ma tipicamente solo poche risultavano effettivamente significative.

http://localhost:8123

SELECT * FROM extracted_data.c5e54033df0a4d17801808da46506300 WHERE l=0

Run (Ctrl+Enter) ✓

timestamp	page_id	page_url	iill	iilr	iirl	iirrl	iirri	iirrililrl	iirrililrili	iirrililrilrilli
-----------	---------	----------	------	------	------	-------	-------	------------	--------------	------------------

Decoders

I decoders, che implementano l'interfaccia *TypeDecoder* e sono collezionati nell'interfaccia *Decoders*, hanno il ruolo di decodificare un qualsiasi tipo di dato nel corretto formato, inoltre il loro metodo principale avrà come valore di ritorno proprio un oggetto del formato specifico a cui è applicato tale metodo, che prende il nome di *decode*.

Quest'ultimo ha il compito, attraverso le *RegeExps*, di normalizzare i dati che arrivano in ingresso e di restituire un dato in formato standard e con assegnazione del tipo a cui corrisponde (es. Un numero di telefono verrà normalizzato secondo una struttura standard e verrà "tipato" come oggetto "Phone", e così via per gli altri tipi di dato). I decoder presenti sono:

- DateDecoders
- ISBNDecoder
- NumberDecoders
- PhoneDecoders
- DimensionalDecoder

Ogni decoder implementa il proprio metodo, chiamato nello stesso modo con l'annotazione di *@Override* per essere utilizzato in maniera centralizzata. Infatti vi è una classe chiamata *UnionDecoder* che ha come scopo proprio quello di unificare tutti i decoders e possiede infatti come elemento, passato come parametro del costruttore, un array di *TypeDecoder*, quindi un array di decoders.

Questa classe presenta sempre il metodo *decode*, il quale scorre, attraverso i vari override, tutti i metodi decode dei decoders presenti nell'array di decoders, fino a trovare il decoder corretto per il tipo di dato passato come input al metodo decode. Così facendo è possibile richiamare una sola volta il metodo decode su un qualsiasi tipo di dato, il valore di ritorno sarà normalizzato e dello stesso tipo dell'oggetto a cui corrisponde.

Value Distances

Questa sezione, contenuta nella cartella "value", si occupa del secondo e forse più importante compito di tutto il progetto: il calcolo della distanza, in termini

di somiglianza, fra due elementi passati come parametri, nel caso in cui entrambi appartengano ad uno stesso tipo di dato (in caso contrario non potranno essere confrontati). Tale valore è compreso fra 0 ed 1 dove 0 indica l'uguaglianza fra i due elementi, mentre l'1 l'assenza di somiglianza. A questo punto è possibile scegliere una soglia sotto la quale due valori sono in matching fra loro, e quindi risultano simili e possono essere aggiunti alla matching list corrispondente.

Ogni tipo di dato avrà il suo specifico metodo di calcolo della distanza (ad esempio due codici ISBN utilizzeranno la *ISBN_DISTANCE*), ognuno implementato in base ai due valori da confrontare, tenendo a mente la differenza insita nel tipo di dato.

Data Type

I valori presenti nella cartella "value" sono tutti i possibili tipi di valori che possono arrivare dai dati estratti nel Database. Alcuni estendono classi già note, mentre altri sono implementati da zero. I valori sono:

- Date
- ISBN
- Number
- Phone
- Dimensional

Normalization (with RegExp)

La normalizzazione attraverso le *RegExp* avviene direttamente nella fase di decodifica, infatti tale linearizzazione dei dati è effettuata in diversi passaggi al fine di produrre un risultato in linea con il tipo di dato che ci si aspetta di ottenere. Se dopo le diverse trasformazioni il dato standardizzato presenta un formato corretto, allora il metodo *decode* ritorna tale valore, altrimenti un semplice *NULL* che suggerisce il mancato riconoscimento della stringa in ingresso e la sua appartenenza ad un formato differente.

Queries - Data extraction

Questa è una delle fasi più importanti di tutto il progetto, infatti è alla base della creazione dei due vettori che verranno poi confrontati per creare una matching list in linea con le specifiche definite. I dati non sono semplicemente caricati sulla macchina che esegue il processo, ma sono contenuti in un server esterno e ciò comporta l'instaurazione di un collegamento sicuro e veloce, per poter gestire le differenti richieste.

Con l'ausilio di alcuni driver, da includere nel *pom.xml* per soddisfare la corrispondente dipendenza, è possibile utilizzare JDBC per l'importazione e la gestione dei dati dal DBMS NoSQL ClickHouse. Tale funzionalità è garantita dalla classe *RetrievingData.java* che verrà implementata come un servizio REST che risponde alla chiamata *"/retrieveData"*.

```
@RequestMapping(value = "/retrieveData", method = RequestMethod.GET)
public Vector<Object> retrieveData(@RequestParam("id") String id, @RequestParam("page_url") String page_url) {
    final String URL = "jdbc:clickhouse://localhost:8123/extracted_data";
    final String USER = "default";
    final String PASSWORD = "";

    Vector<Object> vec = new Vector<>();
    Connection connection = null;
    Statement statement = null;
```

Nel metodo *retrieveData* vengono definiti due parametri che l'utente (o più verosimilmente un altro microservizio) dovrà indicare, che sono l'identificatore della tabella del DMBS (id) e l'url della pagina (page_url) che sarà uno dei due url presenti nella lista di linkage (lista che indica i siti con contenuti simili fra loro).

Successivamente vi sono le definizioni delle variabili, tra cui l'URL del server di ClickHouse, il suo USER e la corrispondente PASSWORD (nel caso in figura è stato instaurato un tunneling e quindi l'host risulta locale). Viene anche inizializzato il vettore che sarà poi passato come output del metodo, il quale risulta essere di tipo *Object*, senza nessuna classificazione dei dati che contiene, la quale avverrà in un secondo momento.

Infatti sarebbe uno spreco di risorse assegnare tale metadata, dal momento che, in ogni caso, la prossima macro fase del progetto ricalcolerà il tipo di dato associato ad ogni oggetto. Vengono inoltre inizializzati connessione e statement.

```
try {
    // register JDBC drive
    Class.forName("ru.yandex.clickhouse.ClickHouseDriver");
    // Open the connection
    connection = DriverManager.getConnection(URL, USER, PASSWORD);
    System.out.println("Connected database successfully");
    // Execute the queries
    statement = connection.createStatement();
    System.out.println("Execute the query");
    String sql = "SELECT * FROM extracted_data." + id + " WHERE page_url = '" + page_url + "'";
    System.out.println(sql);
    ResultSet rs = statement.executeQuery(sql);
    ResultSetMetaData rsmd = rs.getMetaData();

    while (rs.next()) {
        for (int i = 1; i < rsmd.getColumnCount(); i++) {
            System.out.println(i);
            vec.add(rs.getObject(i));
        }
    }
}
```

Ora si esegue la connessione al DBMS e si esegue la query secondo i dati definiti dall'utente (microservizio). Poi la riga corrispondente alla query viene suddivisa in blocchi e memorizzata in un vettore, che sarà proprio il risultato del metodo. L'ausilio dei driver yandex ha reso possibile il funzionamento di ClickHouse con JDBC, il che ha semplificato di molto l'importazione dei dati necessari e dunque della query stessa.

```
} catch (ClassNotFoundException | SQLException e) {
    e.printStackTrace();
} finally {

    // Release resources
    try {
        if (statement != null) {
            statement.close();
        }
    } catch (SQLException throwables) {
        throwables.printStackTrace();
    }
    try {
        if (connection != null) {
            connection.close();
        }
    } catch (SQLException throwables) {
        throwables.printStackTrace();
    }
}

return vec;
```

Qui vengono semplicemente rilasciate tutte le risorse e gestite le varie eccezioni che possono occorrere; infine vi è il return del vettore risultante, come esposto precedentemente. Da notare che in questo caso è stata implementata l'alternativa a microservizi, nella quale il valore di ritorno è proprio il vettore, mentre nel caso HTML sarebbe dovuto essere, appunto, una pagina di questo formato, quindi una *String*, con l'aggiunta del *Model* nell'input del metodo.

Algoritmo di Matching

Dopo aver ottenuto i due vettori di cui abbiamo bisogno, si procede con la fase principale del progetto, cioè il prodotto cartesiano fra i vari elementi dei due vettori, con il calcolo delle distanze tra i due elementi, a patto che questi corrispondano a due formati equivalenti. Tutto ciò viene implementato nella classe *MatchingFinder.java*, anch'essa implementata come un servizio REST.

```
@RequestMapping(value = "/", method = RequestMethod.GET)
public String formThreshold() {

    return "formT";
}
```

La classe è una concatenazione (ma può essere facilmente separata) di due differenti servizi REST. Il primo, che risponde alla chiamata "/", serve per specificare il valore di soglia di matching da utilizzare, così da passarlo come parametro al vero metodo da utilizzare. In questo caso è implementato come input da una pagina HTML, ma in futuro verrà usato da un microservizio a parte per specificare in modo semplice la soglia da usare.

```
@RequestMapping(value = "/application", method = RequestMethod.GET)
public String findMatch(@RequestParam("threshold") String threshold,
                        Vector<Object> input_1, Vector<Object> input_2) {

    Map<Integer, List<Element>> mapResult = new HashMap<>();
    int t = 0;
    Double threshold_final = Double.parseDouble(threshold);
    TypeDecoder[] decoders = {Decoders.ISBN_DECODER, Decoders.NUMBER_DECODER,
                              Decoders.PHONE_DECODER, Decoders.DATE_DECODER};
    UnionDecoder ud = new UnionDecoder(decoders);
```

Com'è possibile notare, in input vi sono la soglia (annotata con *@RequestParam* perchè in questo caso è passata come parametro da una pagina HTML) e i due vettori da confrontare. Da sottolineare che i vettori sono di oggetti di tipo

Object, il valore di ritorno sarà invece una Map con un ID e una lista di oggetti Element.

Tali oggetti hanno come elementi un'etichetta (label) e un campo dati (data). In questo caso, è indicato come return una stringa, infatti il risultato sarà una pagina HTML che stampa proprio la mappa ottenuta, ma nel caso a microservizi sarà, come già detto, una Map. Inoltre viene anche inizializzata una lista di decoders, che servirà al metodo *decode* appartenente alla classe *UnionDecoder* per scorrere tutti i decoder e attraverso l'*@Override* scegliere l'unico che avrà come risultato un valore diverso da NULL (potrebbe anche essere di un formato non supportato, in questo caso sarà considerato sempre NULL).

Tale classe ha il compito, dunque, di unificare tutti i decoders e attraverso una singola chiamata a metodo, poter utilizzare tutti i metodi di tutti i decoders al fine di calcolare non solo il tipo di dato, ma anche di normalizzarlo in un formato standard, comprensibile e semplice da memorizzare.

```
for (int i = 0; i < input_1.size(); i++) {
    for (int j = 0; j < input_2.size(); j++) {

        /* ISBN */
        if (vd.decode(input_1.get(i).toString()) instanceof ISBN &&
            vd.decode(input_2.get(j).toString()) instanceof ISBN) {
            if ((ValueDistances.ISBN_DISTANCE.distance(input_1.get(i).toString(),
                input_2.get(j).toString())) <= threshold_final) {
                List<Element> matchingList = new ArrayList<>();
                matchingList.add(new Element( label: "ISBN", input_1.get(i).toString()));
                matchingList.add(new Element( label: "ISBN", input_2.get(j).toString()));
                mapResult.put(t, matchingList);
                t = t + 1;
            }
        }
    }
}
```

Al fine di garantire il prodotto cartesiano fra i due vettori è stato implementato un semplice nested loop che coinvolge i due vettori in input. Dopo di che iniziano i due if per ogni tipo di dato: il primo serve per verificare che entrambi i campi dei due vettori siano dello stesso tipo, attraverso il metodo decode che ha come valore di ritorno non solo il dato stesso, ma anche il tipo specifico che può essere verificato con instanceof; il secondo, invece, richiama il metodo distance di ValueDistances per il calcolo della distanza, dunque della differenza

e in definitiva del matching, fra i due elementi dei vettori, inserendoli nella lista dei matching finale solo se il valore risulta pari o inferiore della soglia scelta in precedenza.

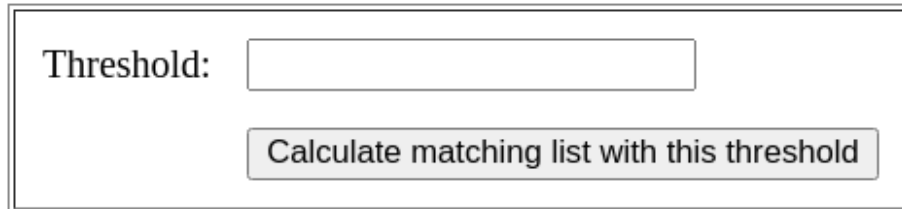
Da notare che l'aggiunta richiede anche la creazione di un oggetto *Element*, per una maggiore chiarezza della mappa finale. Tale oggetto è formato da una *label* che indica il tipo di dato memorizzato ed un campo *data* in cui è salvato il dato in formato standard, dunque normalizzato. Viene poi inserita la lista nella mappa con un ID univoco, che viene incrementato ad ogni aggiunta alla mappa stessa.

Test e Risultati

Il nostro lavoro è parte di un progetto più ampio, perciò quello che abbiamo sviluppato sarà utilizzato da altri microservizi, perciò per testare e mostrare i risultati sono state modificate le classi che implementano i servizi REST per poter interagire con un utente attraverso pagine HTML di richiesta e di risposta.

Nello specifico il test riguarda la scelta della soglia e la stampa della matching list dei valori normalizzati e confrontati attraverso prodotto cartesiano dei due vettori di riferimento, in questo caso codici ISBN (tranne uno che è di tipo "Number" per testare il funzionamento del confronto obbligato solo per lo stesso tipo di dato).

Specify your THRESHOLD



The image shows a web form with a title "Specify your THRESHOLD". Inside the form, there is a label "Threshold:" followed by a text input field. Below the input field is a button with the text "Calculate matching list with this threshold".

Com'è possibile vedere dall'immagine precedente, l'utente è tenuto ad impostare una soglia da 0 a 1 secondo la quale costruire la matching list finale. Senza dubbio maggiore sarà tale valore, più numerosi saranno gli elementi appartenenti alla lista e viceversa.

Ora verrà presentato un esempio di esecuzione; da notare che il metodo richiamato dal servizio REST, è `"/"` per quanto riguarda la scelta della soglia, mentre `"/application"` per quanto riguarda la stampa della lista, ma l'unica da inserire è il primo, visto che il secondo è automaticamente richiamato una volta premuto "Calculate matching list with this threshold".

Specify your THRESHOLD

Threshold:	<input type="text" value="0.8"/>
<input type="button" value="Calculate matching list with this threshold"/>	

Matching list

ID							
0	<table> <tr> <th>Label</th><th>Data</th></tr> <tr> <td>ISBN 978-81-7525-766-5</td><td></td></tr> <tr> <td>ISBN 978-81-7525-210-1</td><td></td></tr> </table>	Label	Data	ISBN 978-81-7525-766-5		ISBN 978-81-7525-210-1	
Label	Data						
ISBN 978-81-7525-766-5							
ISBN 978-81-7525-210-1							
1	<table> <tr> <th>Label</th><th>Data</th></tr> <tr> <td>ISBN 978-81-7525-766-5</td><td></td></tr> <tr> <td>ISBN 285-23-0594-478-5</td><td></td></tr> </table>	Label	Data	ISBN 978-81-7525-766-5		ISBN 285-23-0594-478-5	
Label	Data						
ISBN 978-81-7525-766-5							
ISBN 285-23-0594-478-5							
2	<table> <tr> <th>Label</th><th>Data</th></tr> <tr> <td>ISBN 978-81-7525-766-5</td><td></td></tr> <tr> <td>ISBN 298-99-7525-766-8</td><td></td></tr> </table>	Label	Data	ISBN 978-81-7525-766-5		ISBN 298-99-7525-766-8	
Label	Data						
ISBN 978-81-7525-766-5							
ISBN 298-99-7525-766-8							
3	<table> <tr> <th>Label</th><th>Data</th></tr> <tr> <td>ISBN 978-45-1625-677-2</td><td></td></tr> <tr> <td>ISBN 978-81-7525-210-1</td><td></td></tr> </table>	Label	Data	ISBN 978-45-1625-677-2		ISBN 978-81-7525-210-1	
Label	Data						
ISBN 978-45-1625-677-2							
ISBN 978-81-7525-210-1							
4	<table> <tr> <th>Label</th><th>Data</th></tr> <tr> <td>ISBN 495-20-9823-766-3</td><td></td></tr> <tr> <td>ISBN 298-99-7525-766-8</td><td></td></tr> </table>	Label	Data	ISBN 495-20-9823-766-3		ISBN 298-99-7525-766-8	
Label	Data						
ISBN 495-20-9823-766-3							
ISBN 298-99-7525-766-8							

Com'è possibile notare, la lista finale contiene le coppie di elementi che hanno come soglia di matching un valore inferiore o uguale a 0.8. Questi sono poi divisi nelle rispettive "Label" e "Data", che ne specificano, rispettivamente, le etichette associate al tipo di dato memorizzato e il dato normalizzato secondo lo standard prescelto. Ogni row è poi associata ad un identificatore univoco, che ne semplifica l'accesso.

Specify your THRESHOLD

Threshold:	<input type="text" value="0.5"/>
<input type="button" value="Calculate matching list with this threshold"/>	

Matching list

ID	Label	Data
0	ISBN 978-81-7525-766-5	ISBN 978-81-7525-210-1

In questo secondo esempio è evidente come l'abbassamento della soglia implica un maggior filtro e dunque la necessità di una maggiore somiglianza tra gli elementi e com'è possibile osservare, infatti, i due codici ISBN sono molto simili fra loro.

Conclusioni

I vari task assegnatici all'interno del progetto ci hanno consentito di comprendere e di apprezzare in maniera pratica e diretta gli aspetti studiati all'interno del corso di Big Data, nonché di approfondire e sperimentare tecniche della Data Analytics. Partendo dal dato grezzo e non fruibile siamo riusciti ad identificarne la semantica, a normalizzarlo e renderlo pronto per l'analisi e successivamente al confronto. Infine siamo riusciti a renderlo disponibile e utilizzabile all'interno di una pipeline molto più ampia che è quella del progetto NOAH.

Collateralmente, siamo riusciti ad apprendere la potenza e l'efficacia del DB column-oriented ClickHouse e allo stesso tempo abbiamo ripassato e appreso, rispettivamente, vecchi e nuovi concetti del linguaggio Java. Un ulteriore punto di forza di questo progetto risiede nell'implementazione del servizio REST che aggiunge un fattore polivalente al nostro bagaglio. Infine anche l'attività di refactoring del codice ci ha permesso di sperimentare nuove metodologie per alleggerire il codice e renderlo più leggibile. Dunque, questa esperienza ci ha permesso di visionare una moltitudine di aspetti e mettere mano su svariate tecnologie che oggi rendono più chiara e appassionante la nostra visione del mondo Big Data e aggiungono valore al corso appena concluso.

Bibliografia e sitografia

1. ClickHouse: <https://www.clickhouse.com/company/>
2. IntelliJ IDEA: <https://www.jetbrains.com/idea/features/>
3. Java: https://www.java.com/it/download/help/whatis_java.html
4. ClickHouse to JDBC:
<https://javamana.com/2021/04/20210414213320585k.html>
5. RegExp:
<https://medium.com/@ievgenii.shulitskyi/string-data-normalization-and-similarity-matching-algorithms-4b7b1734798e>
6. Data matching:
<https://www.sciencedirect.com/topics/computer-science/matching-algorithm>
<https://www.carms.ca/the-match/how-it-works/>
7. JDBC: <https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html>