Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Similarità tra stringhe, applicazione dell'algoritmo TLSH a testi di ingegneria

Relatore:
Chiar.mo Prof.
Fabio Vitali

Presentata da:
Giulia Giusti

Sessione II
Anno Accademico 2018/2019

Abstract

La similarity search, ricerca di oggetti simili, è una tecnica fondamentale per l'ottenimento di considerazioni importanti sull'enorme quantità di dati che vengono prodotti ogni giorno. Il progetto riguarda lo sviluppo di un software di ricerca, raggruppamento e classificazione di somiglianza, basato sulla funzione TLSH, di documenti aziendali. A tal proposito, è stata eseguita una ricerca di similarità articolata su tre livelli: sezione, paragrafo e frase.

La funzione TLSH utilizza le migliori tecnologie in termini di efficienza e applicabilità al problema dei Nearest Neighbors: Locality Sensitive Hashing (LSH), Context Triggered Piecewise Hashing (CTPH) e Features Extraction. Queste ultime verificano la proprietà di creazione di digest simili per oggetti simili, fondamentale nella similarity search.

Il sistema del progetto di tesi è composto da due parti distinte: l'ambiente di test della funzione TLSH e il servizio relativo all'API per il confronto tra funzioni LSH.

La valutazione dell'ambiente è stata eseguita attraverso due differenti analisi: quantitativa e qualitativa. La prima con lo scopo di esaminare l'efficienza dell'ambiente in relazione al tempo di esecuzione e alla precisione dei risultati ottenuti. Mentre la valutazione qualitativa riguarda la soddisfazione da parte dell'utente in termini di usabilità del sistema attraverso la somministrazione del questionario SUS (System Usability Scale).

Dall'indagine dei test quantitativi si sono potuti osservare risultati soddisfacenti in efficienza e precisione. Inoltre, i risultati qualitativi si sono rivelati eccellenti con un punteggio SUS molto più elevato del valore medio ottenuto sperimentalmente.

Sviluppi futuri del sistema TLSH potrebbero concernere l'ideazione di un metodo di inserimento dei digest all'interno di una tabella hash, sfruttando al meglio le proprietà di LSH, e l'eliminazione del limite minimo di caratteri richiesti per la creazione di digest.

Indice

Elenco del codice

Elenco delle figure

1. Introduzione

L'obiettivo principale del seguente elaborato è lo sviluppo di un sistema di ricerca, raggruppamento e classificazione di somiglianza basato sulla funzione TLSH al fine di far fronte a problemi legati alla ricerca di testi simili all'interno di una collezione di documenti aziendali di grandi dimensioni.
La tesi contiene un iniziale preambolo teorico finalizzato alla descrizione dello stato dell'arte nella ricerca di oggetti simili e all'esposizione del funzionamento dell'algoritmo TLSH.
Nella seconda parte dell'elaborato viene descritto l'ambiente del progetto svolto e illustrata nel dettaglio la sua implementazione.
Infine, vengono esposte le metodologie di testing utilizzate per la valutazione quantitativa e qualitativa dell'ambiente TLSH e ne vengono analizzati i risultati.

Considerazioni importanti sull'enorme quantità di dati che vengono prodotti ogni giorno possono essere ottenute dalla *similarity search*, cioè la ricerca di elementi simili, come descritto in [Pry06].

I database operazionali delle aziende risultavano molto eterogenei tra loro in termini di coerenza, aggiornamento, completezza e differenze di formati dei dati. Negli anni Novanta l'aumento della produzione dei dati portò alla creazione di database avanzati, chiamati data warehouse, al fine di uniformare le informazioni contenute all'interno dei database operazionali e facilitarne l'analisi e l'elaborazione.

L'ingente quantità di dati in rapida crescita supera la capacità umana di comprendere, osservare e analizzare le complesse informazioni archiviate nei database avanzati. Di conseguenza la ricerca di similarità e le tecniche di data mining diventarono sempre più popolari.

Contrariamente alle query convenzionali basate sulla corrispondenza esatta, usuali per i tradizionali database relazionali, la ricerca di somiglianza trova oggetti che differiscono solo leggermente dall'oggetto query specificato.
I data warehouse sono utilizzati in diversi ambiti, tra i quali: sistemi di riconoscimento biometrico, reti di sensori, database biologici, sistemi automobilistici e aerospaziali, database systems per web data e database multimediali.
Oltre alla quantità dei dati ne aumenta anche la complessità. Ad esempio, database biologici memorizzano informazioni sempre più dettagliate sulle molecole, le applicazioni multimediali archiviano un'enorme quantità di dati complessi costituiti da immagini, audio e video e i documenti HTML forniscono contenuti multimediali incorporati che li rendono molto complicati. Di conseguenza l'analisi di dati complessi pone numerose nuove sfide alla similarity search e alle tecniche di data mining.

Il campo di applicazione della similarity search su cui si concentra tale elaborato è rappresentato dal *Text Mining*, anche conosciuto come *Text Analytics*. In generale il Text Mining consiste nell'applicazione di diverse tecnologie a testi non strutturati al fine di: individuare i principali gruppi tematici, classificare documenti in categorie predefinite, scoprire associazioni nascoste (legami tra argomenti o autori), estrarre informazioni specifiche e concetti per la creazione di ontologie.
La tecnologia di Text Mining è largamente applicata a un'ampia varietà di esigenze del governo, della ricerca e del business. Gli avvocati utilizzano il text mining per l'e-discovery, processo utilizzato nell'ambito dell'informatica forense che consente di individuare dati informatici presenti in archivi digitali al fine di utilizzarli come prove digitali. I governi e i gruppi militari applicano tale tecnica di analisi dei testi per scopi di sicurezza nazionale. I ricercatori scientifici impiegano approcci di estrazione del testo per organizzare grandi documenti di testo al fine di affrontare il problema dei dati non strutturati. Inoltre, per determinare idee comunicate tramite testo viene utilizzato il text mining per la sentiment analysis applicata nel campo del marketing per il posizionamento automatico degli annunci.

La principale applicazione del text mining a cui si dedica tale tesi riguarda la ricerca

di testi simili all'interno di grandi collezioni di documenti.
Si consideri un'azienda che produce un ingente numero di documenti tecnici aventi porzioni di contenuto simili tra loro che necessitano di una traduzione in diverse lingue. La stesura e la traduzione di questi documenti richiede all'azienda un notevole impiego di risorse sia in termini di tempo sia di personale. Quindi risulta necessaria l'ideazione di una tecnologia in grado di sfruttare la similarità intrinseca di questi documenti al fine di abbassare il considerevole costo richiesto all'azienda nelle fasi di stesura e traduzione.
Il problema di ricerca di documenti simili dell'azienda può essere formalizzato in una forma generale chiamata problema dei Nearest Neighbor (NN), problema di ottimizzazione riguardante la ricerca di oggetti di un database che siano simili ad un oggetto dato. Tale problema ha numerose applicazioni nel campo del data mining tra le quali: ricerca di versioni differenti dello stesso documento, individuazione di tutti i documenti correlati che condividono lo stesso oggetto, ricerca di frammenti simili di documenti, analisi del traffico di rete, utilizzo negli analizzatori forensi nell'ambito della sicurezza e ricerca di malware.
Concettualmente il problema dei Nearest Neighbor è facilmente risolvibile effettuando una ricerca lineare all'interno della collezione di documenti; tale soluzione ha però un grande costo computazionale.
L'idea alla base della risoluzione dei limiti della ricerca lineare è quella di collegare la probabilità di collisione dei punti, all'interno dello stesso bucket della tabella hash, con la loro distanza; questa tecnica è detta Locality Sensitive Hashing (LSH).

Al fine di risolvere il problema della ricerca di documenti simili viene utilizzato l'algoritmo TLSH che impiega le migliori tecnologie in termini di efficienza e applicabilità al problema dei Nearest Neighbors: Locality Sensitive Hashing (LSH), Context Triggered Piecewise Hashing (CTPH) e Features Extraction. Queste ultime verificano la proprietà di creazione di digest simili per oggetti simili, fondamentale nella similarity search.

La struttura di questa dissertazione è la seguente. Nel secondo capitolo viene descritto il contesto scientifico tecnologico della *similarity search*.
Nella prima parte del capitolo 2, viene prima formalizzato (sezione 2.1) il problema della

ricerca di similarità attraverso la definizione del problema dei Nearest Neighbors e introdotta la terminologia di base in tale ambito di ricerca.
Nella seconda parte del capitolo 2 vengono illustrati tre importanti aspetti riguardanti la *similarity search*. Nella sezione 2.2 vengono proposti tre diversi tipi di calcolo degli hash al fine di identificare la tecnologia in grado di garantire valori hash simili per oggetti simili. Nella sezione 2.3 vengono descritte due tecniche di ricerca di oggetti simili al fine di individuare quella più efficiente in termini di costo computazionale. Inoltre, nella sezione 2.4 del capitolo vengono introdotte alcune metodologie per la selezione di caratteristiche dai dati. L'identificazione di una giusto metodo di features extraction è fondamentale per l'ottenimento di una funzione di ricerca di similarità precisa. Infine, nell'ultima sezione del capitolo (2.5) viene descritto il calcolo della distanza di Levenshtein che viene successivamente utilizzata nell'implementazione del sistema del progetto di tesi.

Nel terzo capitolo viene illustrato nel dettaglio il funzionamento dell'algoritmo TLSH. La descrizione di tale funzionamento è suddivisa essenzialmente in due parti. La prima parte, contenuta nella sezione 3.1, riguarda la costruzione del digest TLSH. La seconda parte, contenuta nella sezione 3.2, si riferisce al metodo utilizzato per il calcolo della distanza tra i digest TLSH.

Nel quarto capitolo viene illustrata la struttura generale del sistema basato su TLSH, l'interfaccia dell'ambiente, le funzionalità principali e le applicazioni della ricerca degli oggetti simili in vari ambiti. Inoltre, per ogni funzionalità viene descritta l'utilità e la modalità d'uso da parte dell'utente.

Nel quinto capitolo viene descritta nel dettaglio l'implementazione del sistema ponendo particolare attenzione sulle scelte implementative adottate e sulla loro motivazione. Tale descrizione dell'implementazione è suddivisa in due parti. La prima parte, contenuta nella sezione 5.1, si riferisce all'implementazione dell'ambiente di test TLSH. La seconda parte, reperibile nella sezione 5.2, riguarda la realizzazione dell'API per il confronto tra funzioni LSH.

Nel sesto capitolo vengono riportati i risultati dei test effettuati sul sistema. Quest'ultimi sono di due tipi: qualitativi e quantitativi. I primi sono volti ad identificare il livello di soddisfazione dell'utente, mentre il secondo tipo valuta le performance oggettive del sistema. Inoltre, per ogni test vengono analizzati i risultati ed illustrate le scelte implementative che potrebbero aver causato eventuali anomalie nei dati ottenuti.

Infine, nell'ultimo capitolo, vengono riportate le conclusioni ottenute dall'analisi dei test del sistema e conseguentemente vengono ideati alcuni possibili sviluppi futuri dell'ambiente di TLSH.

2. Il problema del Similarity Search

Il principale problema affrontato da questo elaborato di tesi riguarda la ricerca di file simili. Al fine di trovare la migliore soluzione a tale quesito è stata effettuata una ricerca volta a verificare quali tecnologie siano state adottate precedentemente all'ideazione della funzione TLSH e analizzare i miglioramenti apportati dalle tecniche utilizzate da questa funzione.
In questo capitolo viene inizialmente illustrata la formalizzazione del problema riguardante la ricerca di oggetti simili e in seguito vengono introdotte le tecnologie utilizzate in tale ambito di ricerca; per ognuna di esse viene descritto il funzionamento, i limiti e le eventuali soluzioni esistenti.

2.1 Formalizzazione del problema

Nella seguente sezione viene descritto formalmente il problema dei Nearest Neighbors. Inoltre viene introdotta la terminologia di riferimento adottata in tale ambito al fine di descrivere il significato dei principali termini utilizzati e fissare un glossario comune.

2.1.1 Problema ricerca dei Nearest Neighbors (NN)

Dato un punto q, chiamato *query point*, si desidera trovare i punti di un grande database (nel nostro caso, una grande collezione di documenti) che sono più vicini al punto q. Quindi, per qualsiasi query point, si vuole essere in grado di restituire i punti ad esso più vicini, chiamati *nearest neighbors*, con un'alta probabilità.
In tale studio il problema sorge quando si devono considerare un ingente numero di coppie di elementi per calcolare il grado di somiglianza. [CM15]
Il problema dei Nearest Neighbors si presenta in molti ambiti del data mining, campo interdisciplinare che si occupa dell'analisi e della modellazione di grandi volumi di dati

attraverso l'uso di algoritmi per l'estrazione di informazioni aggiuntive interessanti e non note a priori. I suoi principali ambiti di applicazione sono:

- Ricerca di versioni differenti dello stesso documento;
- Ricerca di tutti i documenti correlati che condividono lo stesso oggetto (es: immagine);
- Ricerca di frammenti simili di documenti;
- Analisi del traffico di rete;
- Ricerca eseguita da analizzatori forensi nell'ambito della sicurezza e dell'analisi forense;
- Ricerca di malware.

Alcuni dei casi d'uso sopra elencati vengono descritti più precisamente in 4.1.

2.1.2 Terminologia

La terminologia illustrata nella seguente sezione fu ideata nel 2014 dal National Institute for Standard and Tecnologies (NIST) al fine di unificare il glossario riguardo lo studio del problema dei Nearest Neighbors e facilitare la discussione e la ricerca in tale campo scientifico.

Esistono molti modi per interpretare la similarità tra stringhe. Si considerino ad esempio le stringhe: "ababa" e "cdcdc"; esse possono essere considerate simili perché entrambe contengono cinque caratteri con valori alternati, oppure le medesime possono non essere considerate simili perché non presentano nessun carattere in comune.
Al fine di risolvere questa ambiguità, la similarità tra due oggetti viene definita in termini di *features*, o caratteristiche, che rappresentano le proprietà interessanti degli oggetti per il metodo di confronto. [Bre+14]

Features

Le features rappresentano elementi base mediante i quali l'oggetto digitale viene confrontato con altri oggetti. Il confronto tra due features restituisce sempre un valore binario (0 o 1) che indica il successo o il fallimento del confronto, questo perché la feature è definita come l'unità base presa in considerazione dall'algoritmo e i match parziali non sono ammessi.
Ogni algoritmo per la ricerca di oggetti simili deve definire la struttura delle features utilizzate e il metodo attraverso il quale esse vengono determinate.

Feature set

Per feature set si intende l'insieme di tutte le features associate ad un singolo oggetto digitale.

Similarità

La similarità tra due artefatti, misurata attraverso un particolare algoritmo, è definita come una funzione crescente monotona del numero dei matching delle features che hanno avuto successo.

Similarity digest

Il similarity digest, anche chiamato valore hash o semplicemente digest, è una rappresentazione del feature set dell'oggetto digitale che è adatto al confronto con altri digest creati con lo stesso algoritmo.
Le funzioni di ricerca dei Nearest Neighbors devono necessariamente generare digest simili per oggetti simili. Una tecnica di calcolo di valori hash che non verifica la proprietà di digest simili per oggetti simili non può essere applicata alla similarity search. Nella prossima sezione vengono approfondite due tecnologie che presentano tale problema (vedi sottosezioni: 2.2.1, 2.2.2).

Complessità computazionale

La complessità computazionale si occupa della valutazione del costo degli algoritmi in termini di risorse di calcolo:

- $time_n()$: tempo di elaborazione
- $space_n()$: quantità di memoria utilizzata

Individuare con esattezza la funzione $time_n()$ è molto complicato. Tuttavia, nella maggior parte dei casi è sufficiente stabilire il comportamento asintotico della funzione quando le dimensioni dell'input tendono ad infinito.
Un algoritmo ha complessità O(f(n)) se esistono opportune costanti a, b, n' tali che $\forall n > n'$:

$$time_n() \leq a * f(n) + b$$

O(f(n)) si dice limite superiore al comportamento asintotico della funzione.
Relativamente a tale costo computazionale viene valutata l'efficienza dell'algoritmo.

2.2 Tecnologie di calcolo del similarity digest

Nella seguente sezione vengono illustrate alcune tecniche utilizzate per il calcolo dei valori hash del similarity digest, usato per rappresentare le features su cui si effettua il confronto di similarità. In particolare, le prime due risultano non essere adeguate per la ricerca di oggetti simili poiché non verificano il requisito fondamentale dei digest descritto precedentemente, ovvero digest simili per stringhe simili. Infine viene esposta una tecnica migliorativa che rispetta il requisito sui digest, chiamata Context Triggered Piecewise Hashing (CTPH), sulla quale si basa la funzione TLSH.

2.2.1 Hash crittografici

Le funzioni hash crittografiche, come MD5 e SHA-1, fanno parte di una classe speciale di funzioni hash che dispongono di alcune proprietà che le rendono applicabili nell'ambito della sicurezza informatica.
Le funzioni che calcolano gli hash crittografici hanno le seguenti proprietà:

- La dimensione dell'input è arbitraria;
- La dimensione del valore hash in output è fissa;
- La funzione è di tipo deterministico, quindi più esecuzioni della funzione sullo stesso input devono restituire il medesimo valore hash;
- La funzione è semplice e veloce da calcolare;
- Dato un valore hash deve essere molto difficile o quasi impossibile trovare l'oggetto che ha prodotto tale valore, questa caratteristica è denominata *hiding*;
- Dato un qualunque valore in input x_1 è praticamente impossibile trovare un altro valore di input x_2 diverso da x_1 per cui la funzione restituisce lo stesso valore hash, tale proprietà è nota come *collision free*;
- Oggetti simili vengono mappati in valori hash molto diversi, quindi se viene modificato anche un solo bit dell'oggetto in input allora l'output risulterà completamente diverso. Tale peculiarità è detta *collision avoidance*.

Le caratteristiche appena descritte sono largamente utilizzate per due differenti impieghi: l'analisi dell'uguaglianza tra oggetti e il controllo dell'integrità. Nel primo caso, vengono identificati come uguali gli oggetti che producono lo stesso valore hash. Nel secondo caso, se l'integrità di un oggetto viene violata attraverso una modifica allora il calcolo del valore hash produce un valore diverso rispetto al precedente. [HBB16][Kor06]

Gli hash crittografici non risultano adeguati agli obiettivi sopra prefissati poiché, per la proprietà di collision avoidance, due oggetti simili possiedono valori hash completamente diversi e questo viola il requisito fondamentale dei digest. [MH17]

2.2.2 Block-Based Hashing

La tecnica chiamata Block-Based Hashing [HBB16], anche conosciuta come Piecewise Hashing, venne introdotta nel 2006 al fine di rilassare la proprietà di collision avoidance delle funzioni hash crittografiche. Secondo tale tecnica l'oggetto in input viene suddiviso

in blocchi di dimensione fissa e per ogni blocco viene calcolato il digest mediante una funzione hash crittografica. Il valore hash complessivo dell'oggetto si ottiene concatenando i valori hash di tutti i blocchi.

Si considerino gli oggetti come stringhe di caratteri. La tecnica di Block-Based Hashing verifica il requisito fondamentale dei digest nel caso di sostituzione di un carattere con un altro, poiché in tale situazione viene alterato solo l'hash del blocco contenente il valore modificato e quindi solamente una parte del digest complessivo dell'oggetto risulta modificato.
Tuttavia i Block-Based Algorithm soffrono del problema dell'allineamento secondo il quale operazioni di inserimento o rimozione anche di un solo carattere all'inizio della sequenza rappresentante l'oggetto in input hanno effetti sul contenuto dei blocchi successivi e quindi il valore hash complessivo si rivela completamente diverso.

A causa di questo problema di allineamento anche i Block-Based Algorithm non possono essere utilizzati per la ricerca di similarità poiché viene violato il requisito fondamentale dei digest.

2.2.3 Context Triggered Piecewise Hashing (CTPH)

Al fine di risolvere il problema di allineamento della tecnica di Block-Based Hashing, la quale utilizza offset fissi per determinare la suddivisione dell'input in blocchi, fu ideata una tecnica migliorativa chiamata CTPH [Kor06].
La tecnica CTPH, illustrata in questa sotto sezione, utilizza un metodo di suddivisione nel quale gli offset vengono generati da un rolling hash. L'utilizzo di un rolling hash risulta essere fondamentale per ottenere digest simili per oggetti simili che è indispensabile per l'utilizzo di questi hash nel campo del similarity search.

Rolling Hash

Gli algoritmi di rolling hash producono in output un valore pseudocasuale basato unicamente sul contesto corrente dell'input; in particolare, mantengono uno stato basato solo sugli ultimi byte di quest'ultimo.

Si considerino gli oggetti come stringhe di caratteri. Inoltre, si supponga di avere n caratteri e si immagini che i primi i caratteri siano rappresentati da un byte b_i, quindi l'input consiste nei seguenti byte:

$$b_1, b_2, ..., b_n$$

L'input viene esaminato attraverso una finestra scorrevole di dimensione s.
Per ogni posizione p dell'input lo stato del rolling hash dipende solo dagli s byte dell'oggetto considerati in tale iterazione. Quindi il valore del rolling hash r può essere espresso in funzione di questi ultimi byte nel seguente modo:

$$r_p = F(b_p, b_{p-1}, b_{p-2}, ..., b_{p-s})$$

dove la funzione di rolling hash F è costruita in modo che sia possibile rimuovere l'influenza di uno dei termini. Conseguentemente, dato r_p è possibile calcolare r_{p+1} rimuovendo l'influenza del termine b_{p-s} e aggiungendo l'influenza di b_{p+1}, nel seguente modo:

$$r_{p+1} = r_p - X(b_{p-s}) + Y(b_{p+1}) = F(b_{p+1}, b_p, b_{p-1}, ..., b_{(p-s)+1})$$

dove $X(b_{p-s})$ rappresenta l'influenza del termine b_{p-s}, mentre $Y(b_{p+1})$ rappresenta l'influenza di b_{p+1}.
In questo modo ad ogni spostamento della finestra scorrevole di dimensione s risulta semplice ricalcolare il valore del rolling hash da restituire. [Kor06]

Combinazione degli algoritmi di hash in CTPH

Diversamente dai tradizionali algoritmi di Piecewise Hashing, che utilizzano offset fissi per determinare la suddivisione dell'input in blocchi, CTPH utilizza un rolling hash.
Una finestra di dimensione fissa s viene spostata sull'oggetto in input, ad ogni spostamento di tale finestra la funzione di rolling hash restituisce un valore basato sugli s caratteri considerati. Il valore restituito dal rolling hash viene usato come offset e un algoritmo di hash tradizionale viene applicato al blocco di input delimitato da tale offset. Infine il valore hash del blocco viene memorizzato nel digest di CTPH.
In questo modo ogni valore hash memorizzato nel digest complessivo dipende unicamente

da una parte dell'input e le modifiche a quest'ultimo comportano alterazioni localizzate nella firma CTPH. Quindi se un byte di input viene modificato allora la maggior parte della firma di CTPH rimane la medesima e gli oggetti noti modificati possono essere associati a oggetti conosciuti grazie al loro digest CTPH simile.
Pertanto CTPH verifica il requisito di digest simili per oggetti simili e quindi risulta essere una tecnologia adeguata per la ricerca dei Nearest Neighbors. [Kor06]

2.3 Tecniche di ricerca di oggetti simili

Nella seguente sezione vengono descritte due tecniche di ricerca di oggetti simili al fine di individuare quella più efficiente in termini di costo computazionale.

2.3.1 Approximate Matching

Approximate Matching [Bre+14] è una tecnica che ha come obiettivo quello di identificare la similarità tra due oggetti digitali creando valori hash confrontabili.
In questo contesto per oggetti o artefatti digitali si intende una qualsiasi sequenza di bit.

Tipi di Approximate Matching

Differenti metodi di Approximate Matching possono operare su diversi livelli di astrazione. In relazione al livello di astrazione si possono distinguere i seguenti tipi di matching:

- Bytewise matching
  Questi metodi si basano unicamente sulla sequenza di byte che compongono un oggetto digitale, senza considerare la struttura dei dati o il significato dei byte qualora vengano interpretati in modo appropriato.

- Syntactic matching
  Questi metodi considerano la struttura interna dell'oggetto digitale (es: struttura dei pacchetti TCP). Il calcolo di similarità è specifico per una particolare classe di oggetti che condividono una codifica ma non richiedono nessuna interpretazione del contenuto per produrre risultati utili.

- Semantic matching
  Questi metodi utilizzano attributi di contesto dell'oggetto digitale per interpretare quest'ultimo in una maniera che sia più vicina possibile alla percezione umana.
  Il semantic matching restituisce risultati maggiormente specifici rispetto ai precedenti metodi, tuttavia è computazionalmente più costoso perché richiede un'analisi più approfondita dei dati.

Per la risoluzione del problema cardine dell'elaborato di tesi si è principalmente interessati al livello più basso, cioè il Bytewise Matching. Tali metodi godono di grande applicabilità poiché assumono che gli oggetti percepiti dagli umani come simili abbiano una codifica a livello di byte simile.

Tecniche di Bytewise Approximate Matching

Gli algorimi di Bytewise Approximate Matching [Bre+14] lavorano in due fasi. La prima fase calcola il digest dell'oggetto, mentre la seconda si occupa della produzione del *similarity score*.

Gli algoritmi di Bytewise Approximate Matching richiedono l'utilizzo di almeno due funzioni:

- Feature extraction function
  Questa funzione ha il compito di identificare ed estrarre le features dall'oggetto considerato.
  Uno studio più approfondito riguardo le tecniche di Feature Extraction viene effettuato in 2.4

- Similarity function
  Questa funzione confronta i digest dei due oggetti e restituisce il *similarity score*, cioè il valore che ne descrive il grado di similarità.
  Il *similarity score* solitamente è un numero compreso tra 0 e 1, dove 0 indica nessuna similarità e 1 alta similarità.

Problema dell'Approximate Matching

Il grande limite delle funzioni di Approximate Matching è rappresentato dalla ricerca della similarità, in essa si vogliono trovare oggetti simili che hanno un certo grado di condivisione del contenuto, prendendo in considerazione unicamente i loro digest.
Molti algoritmi di Approximate Matching utilizzano un metodo di forza bruta per la ricerca della similarità, questo significa che ogni oggetto del sistema bersaglio viene comparato con tutti gli oggetti del sistema di riferimento. Questo processo ha un grande costo computazionale per i dataset molto grandi. La complessità della ricerca è O(r n) dove n è il numero di digest del sistema bersaglio e r è il numero di digest del sistema di riferimento.

2.3.2 Locality Sensitive Hashing

Nel 2014 fu proposta una nuova tecnica per la ricerca degli oggetti simili, denominata Locality Sensitive Hashing [SC08][CM15], finalizzata al miglioramento del costo computazionale richiesto dall'Approximate Matching.
Dati due punti, l'idea di base di LSH è quella di collegare la probabilità di collisione dei punti, all'interno dello stesso bucket della tabella hash, con la loro distanza. Se due punti sono vicini allora la probabilità che essi collidano è maggiore; quest'ultima diminuisce in modo proporzionale alla crescita della distanza tra i punti considerati.

Descrizione formale dell'idea di LSH

Nella descrizione della tecnica di LSH gli oggetti da analizzare vengono rappresentati come punti all'interno di uno spazio vettoriale di dimensione n. In altre parole, un oggetto viene visto come un vettore n-dimensionale che può essere rappresentato in un sistema cartesiano con n assi ortogonali.
Si definisce con il termine proiezione una funzione che mappa i punti da uno spazio di grandi dimensioni in un sottospazio di piccole dimensioni.
Considerati due punti p e q, se essi risultano vicini tra loro allora rimangono tali anche in seguito ad una proiezione.
Per calcolare insiemi di oggetti simili mediante LSH viene eseguito il seguente processo:

1. Creazione di diverse proiezioni con direzioni differenti e memorizzazione, per ciascuna di esse, dell'elenco dei punti che si trovano vicini.

2. Ricerca dei punti che si trovano vicini in più di una proiezione e rappresentazione di tali punti in insiemi.

I punti che vengono rappresentati all'interno dello stesso insieme dall'operazione di ricerca vengono considerati simili tra loro. [SC08] [LRU14]

Metafora della sfera

In molti articoli [SC08] [CM15], per spiegare la tecnica del Locality Sensitive Hashing, viene introdotta la metafora della sfera. In tale allegoria viene considerato un insieme di punti situato sulla figura tridimensionale della sfera. In seguito, vengono analizzate le posizioni reciproche assunte da tali punti quando la sfera viene rappresentata su uno spazio bidimensionale.
Due punti che risultano essere vicini su una sfera rimangono tali anche quando la sfera viene proiettata su uno spazio bidimensionale e questo vale indipendentemente da come viene ruotata la sfera. Al contrario, due punti che sono lontani sulla sfera possono apparire vicini in seguito ad una proiezione su uno spazio bidimensionale. Tuttavia, nella maggior parte delle orientazioni, questi due punti risultano comunque lontani.

Funzionamento pratico di LSH

Si consideri la metafora della sfera sulla quale si basano le immagine sotto riportate.
Come descritto precedentemente, l'obiettivo è quello di generare codici hash simili per elementi vicini. Il metodo utilizzato a tale fine viene sintetizzato come segue:

1. Viene considerato uno spazio vettoriale di dimensione 3, dunque dove ogni punto è formato da un vettore di dimensione 3 di numeri booleani (0/1).

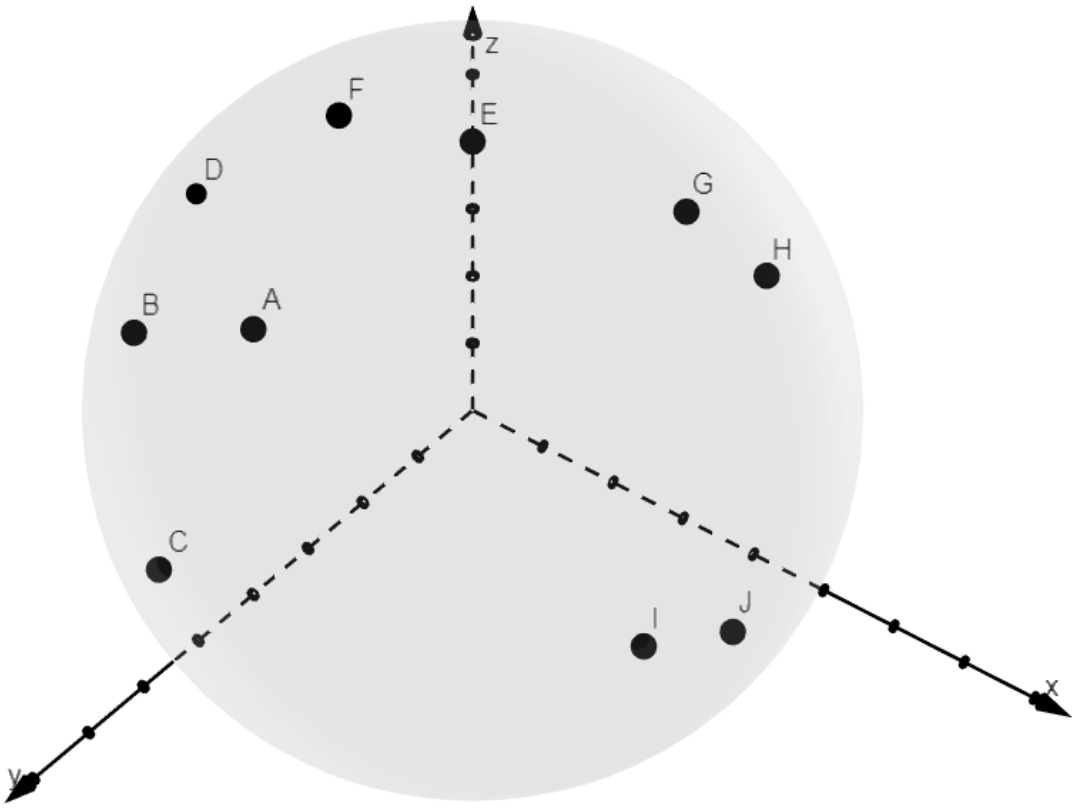

Figura 2.1: Grafico dello spazio vettoriale

2. Vengono generati, in maniera casuale, tre iperpiani aventi differenti angolazioni. Un iperpiano suddivide lo spazio considerato in due diversi sottospazi che chiameremo arbitrariamente sottospazio positivo (bit a 1) e sottospazio negativo (bit a 0).

   (a) Generazione dell'iperpiano h1:

   - Tutti i punti sopra ad h1 hanno come primo bit del digest il valore 0;
   - Tutti i punti sotto ad h1 hanno come primo bit del digest il valore 1.

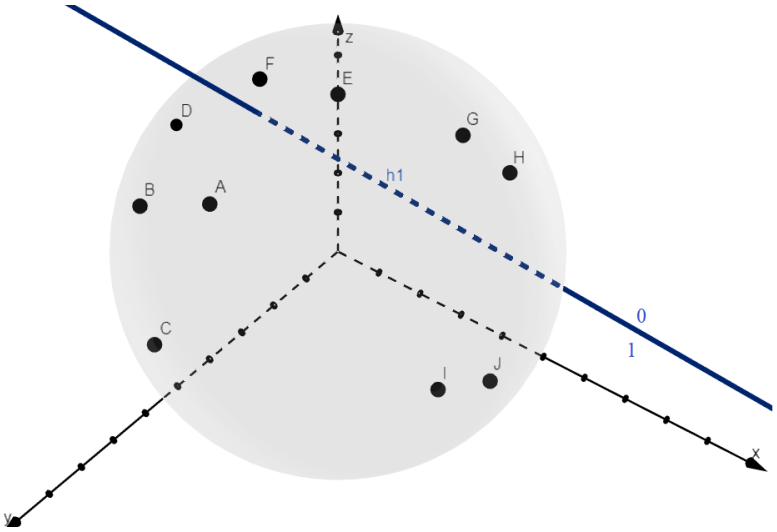

Figura 2.2: Grafico dell'introduzione del primo iperpiano h1 nello spazio vettoriale

(b) Generazione dell'iperpiano h2:

- Tutti i punti sopra ad h2 hanno come primo bit del digest il valore 0;
- Tutti i punti sotto ad h2 hanno come primo bit del digest il valore 1.

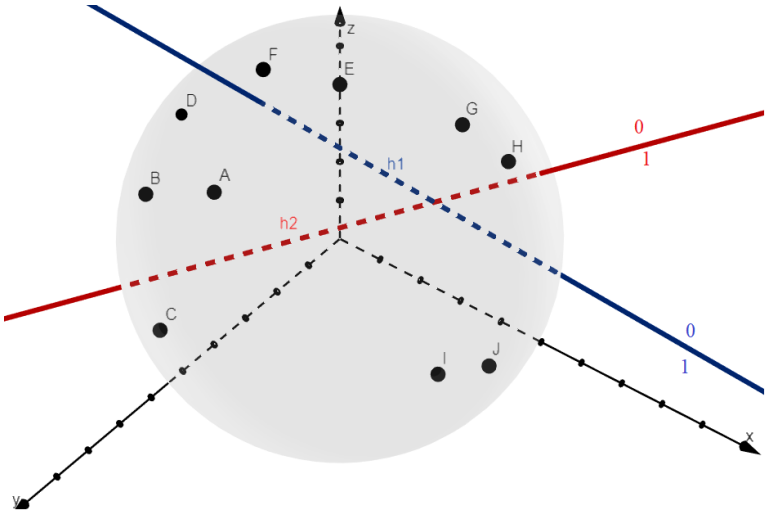

Figura 2.3: Grafico dell'introduzione del secondo iperpiano h2 nello spazio vettoriale

(c) Generazione dell'iperpiano h3:

- Tutti i punti sopra ad h3 hanno come primo bit del digest il valore 0;
- Tutti i punti sotto ad h3 hanno come primo bit del digest il valore 1.

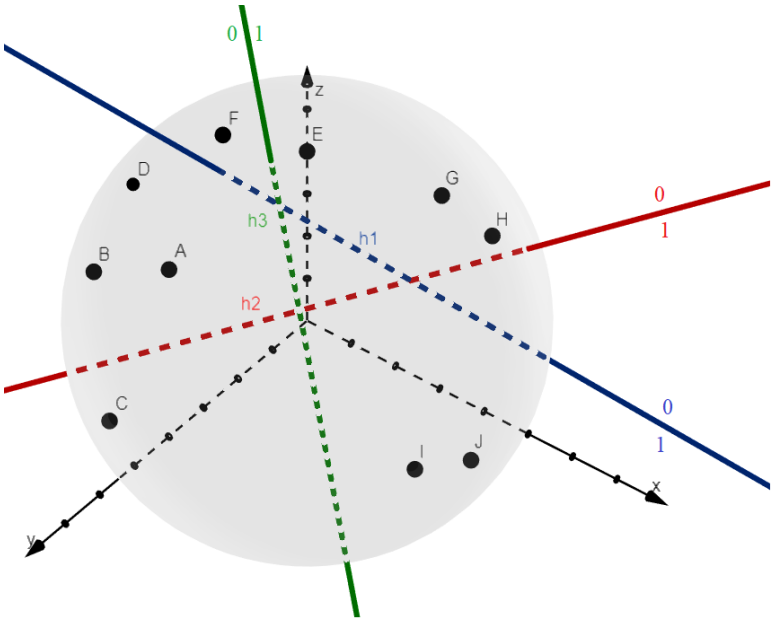

Figura 2.4: Grafico dell'introduzione del terzo iperpiano h3 nello spazio vettoriale

3. Si costruiscono i digest in base alla regione in cui è posizionato il punto.
   I punti che che si trovano nella stessa regione hanno lo stesso digest e vengono mappati nello stesso bucket della tabella hash.
   Si ottiene la seguente tabella hash:

   - Bucket 000: F
   - Bucket 001: G,H,E
   - Bucket 011: vuoto
   - Bucket 111: I,J
   - Bucket 110: C
   - Bucket 100: A,B,D
   - Bucket 101: vuoto

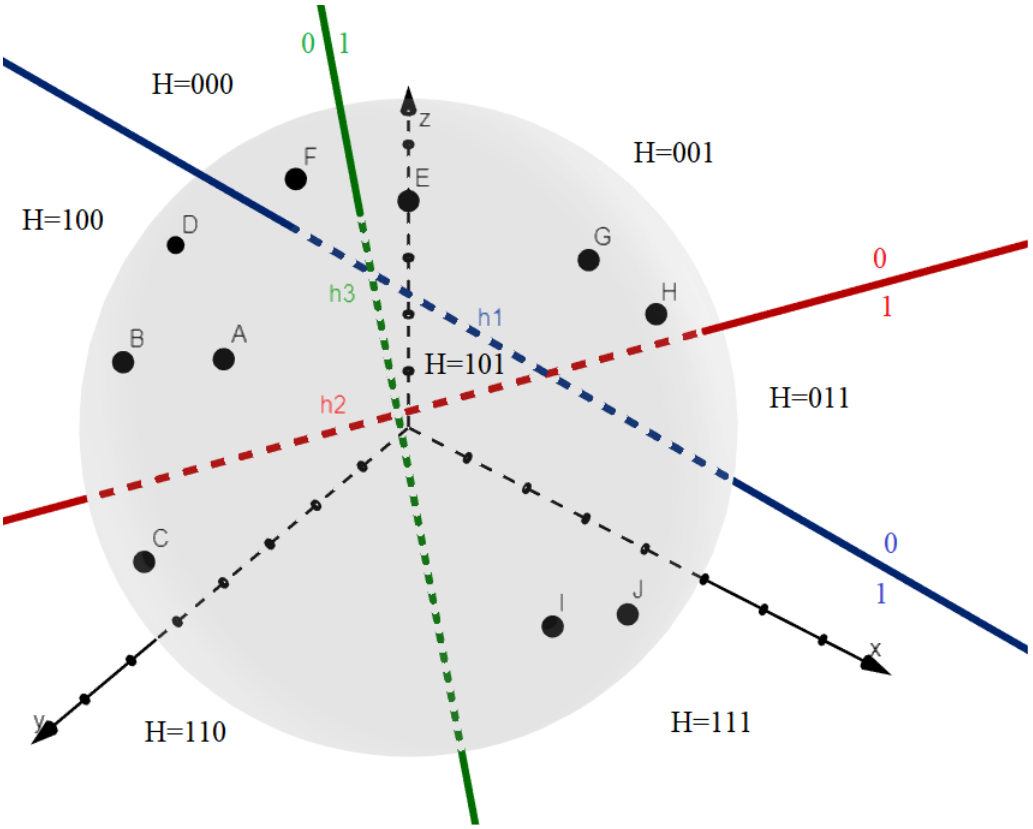

Figura 2.5: Descrizione dei digest in base alla posizione degli iperpiani

4. Vengono confrontati i diversi punti che si trovano nello stesso bucket.
   Esempio: All'interno del bucket con hash code 100, A viene comparato con B e D e si ottengono i seguenti risultati:
   - B risulta essere simile ad A perché è molto vicino ad esso.
   - D è un falso positivo perché è lontano da A ma gli iperpiani scelti li mappano nella stessa regione e di conseguenza viene eliminato.
5. Viene ripetuto il processo con altri 3 iperpiani creando un'altra tabella hash.

L'ottenimento di insiemi di elementi simili avviene iterando il processo e successivamente considerando i punti che vengono mappati più volte nello stesso bucket.

Dall'analisi dei risultati ottenuti si definiscono falsi positivi e falsi negativi i risultati aventi le seguenti caratteristiche:

- Falsi negativi in LSH: si presentano quando due elementi a e b sono molto simili ma in nessuna ripetizione del processo precedentemente illustrato essi compaiono nello stesso bucket.
- Falsi positivi in LSH: si presentano quando a e b non sono molto simili ma in alcune delle iterazioni, del processo precedentemente illustrato, essi compaiono nello stesso bucket.

Il numero di iterazioni necessarie per evitare falsi positivi o negativi non è noto a priori poiché dipende fortemente dalla distribuzione dei punti sul piano.
Al fine di confrontare la tecnica appena illustrata con l'Approximate Matching, è necessario studiarne l'intrinseca complessità. A questo proposito si definiscono i seguenti valori:

- N $\rightarrow$ punti nello spazio considerato
- D $\rightarrow$ dimensione del vettore di ogni punto che corrisponde con la dimensione dello spazio vettoriale
- K $\rightarrow$ numero di iperpiani
- DK $\rightarrow$ costo per trovare il bucket di ogni elemento
- D$\frac{N}{2^K}$ $\rightarrow$ costo di comparazione perché con $\frac{N}{2^K}$ si indica il numero medio di punti nello stesso bucket
- L $\rightarrow$ numero di ripetizioni del processo descritto precedentemente

Quindi il costo di LSH risulta essere: L(DK+D$\frac{N}{2^K}$) che equivale a O(log N) se consideriamo D$\sim$logN

Differenza tra LSH e Approximate Matching

Le tecniche di LSH e Approximate Matching differiscono per due principali caratteristiche: la prima più di carattere concettuale, mentre la seconda basata sul costo computazionale.
Infatti l'idea dietro LSH è quella di mappare, con alta probabilità, oggetti simili all'interno degli stessi bucket di una tabella hash. Diversamente l'idea cardine delle tecniche di Approximate Matching è quella di restituire similarity digest che siano comparabili e ottenere oggetti simili in seguito al confronto di tali digest.
Per quanto riguarda la differenza relativa al costo computazionale, nel caso dell'Approximate Matching è necessaria una scansione lineare e quindi il costo risulta essere nell'ordine di O(N). Diversamente, in LSH si ha un costo nell'ordine O(log N).
In conclusione, LSH risulta essere una tecnica più efficiente nella ricerca dei Nearest Neighbors rispetto all'Approximate Matching. [HBB16] [MH17]

2.4 Metodologie di features extraction

Come descritto precedentemente in 2.1.2, l'idea generale alla base di qualsiasi schema di somiglianza tra oggetti è quella di selezionare le caratteristiche di un oggetto e confrontarle con le caratteristiche selezionate da altri oggetti. L'insieme delle caratteristiche di un oggetto può essere visto come il fingerprint di quest'ultimo.

Lo scopo di questa sezione è quello di presentare un'introduzione superficiale delle principali tecniche di features extraction [Rou10] poiché TLSH, tema prevalente dell'elaborato, utilizza tecniche di tale tipo per estrarre le features dagli oggetti che prende in input.

2.4.1 Indice di Jaccard

Il primo approccio alla feature extraction è rappresentato dal lavoro del biologo svizzero Paul Jaccard nel 1902; egli suggerì di esprimere la similarità tra due insiemi finiti come il rapporto tra la dimensione della loro intersezione e la dimensione della loro unione.
Si considerino due oggetti A e B, l'indice J ipotizzato da Jaccard tra A e B è definito

dalla seguente formula:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

A causa della sua semplicità, esso venne sostituito nel 1981 dalla tecnica chiamata Rabin Fingerprinting.

2.4.2 Rabin Fingerprinting

Lo schema proposto da Rabin è basato su polinomi random e ha come obiettivo quello di fornire un algoritmo molto semplice di string matching e una procedura di controllo dell'integrità dei dati.
La prima applicazione del fingerprint di Rabin riguarda il rilevamento efficiente di oggetti identici.
Negli anni '90 l'idea di Rabin venne utilizzata nel contesto della ricerca degli oggetti simili. L'idea di base, definita come anchoring o chuncking, era quella di utilizzare i fingerprint di Rabin su una finestra scorrevole di dimensioni fisse per suddividere l'oggetto in input in blocchi. Tale suddivisone avviene nel seguente modo:

- Un valore di hash h viene calcolato per ogni finestra di dimensione w;
- Il valore h viene diviso per una costante c e il resto viene confrontato con un'altra costante m. Nel caso in cui i due valori risultino uguali, ovvero $m \equiv h \ mod \ c$, allora i dati della finestra vengono dichiarati come inizio di un chunk o anchor e la finestra scorrevole viene spostata di una posizione.

Il processo appena illustrato viene ripetuto fino al raggiungimento della fine dei dati dell'oggetto.
Il valore c è in genere una potenza di due ($c = 2k$) e m è un numero fisso compreso tra 0 e $c - 1$.
Una volta determinato l'ancoraggio di base tale tecnica può essere utilizzata in diversi modi per selezionare le caratteristiche degli oggetti. [Rou10]

Il modello ideato da Rabin applicato al problema di ricerca di Nearest Neighbors presenta due grandi problemi: bassa copertura ed elevato numero di falsi positivi. Que-

st'ultimi possono essere entrambi ricondotti al fatto che i dati sottostanti possono essere significativamente diversi in relazione al loro contenuto informativo.

2.4.3 Non-Rabin Fingerprinting

Nel 2017 furono introdotte nuove tecniche di Features Extraction volte a superare i limiti della tecnologia ideata da Rabin. Il modello su cui queste nuove tecniche si basano è detto Non-Rabin Fingerprinting [Rou10]. In tale modello viene definita *caratteristica di un oggetto* una sequenza di bit dell'oggetto considerato, quindi vengono considerati i dati binari come entità sintattiche senza tentare di analizzarli o interpretarli. Questo approccio presenta evidenti limiti, ma è motivato dalla necessità di sviluppare un metodo generico in grado di filtrare grandi quantità di dati.
Al fine di migliorare i risultati del Rabin Fingerprinting vengono aggiunte le seguenti proprietà:

- nuovo schema di selezione delle caratteristiche chiamate *Statistically-Improbable Features* (SIF), il quale fornisce un'identificazione più affidabile delle caratteristiche e offre una copertura maggiore;
- ricerca delle caratteristiche intrinsecamente deboli che porta ad una riduzione dei falsi positivi;
- nuova misura scalabile di somiglianza che permette il confronto efficace tra oggetti di dimensioni arbitrarie.

Avendo una riduzione dei falsi positivi e una copertura maggiore tale tecnica di features extraction risulta essere più adeguata per la ricerca degli oggetti simili.

2.5 Distanza di Levenshtein

In questa sezione viene descritta la distanza di Levenshtein, misura finalizzata al calcolo della similarità tra stringhe. Tale distanza viene utilizzata nell'implementazione del sistema TLSH come metrica oggettiva di valutazione dell'algoritmo al fine di testare la precisione dei risultati.

Lo scienziato russo Vladimir Levenshtein nel 1965 introdusse la distanza di Levenshtein o distanza di edit. Quest'ultima rappresenta una misura finalizzata al calcolo della similarità tra stringhe.
La distanza di Levenshtein [JM18] è definita come il minimo numero di operazioni elementari necessarie per trasformare una stringa in un'altra stringa. Le operazioni elementari considerate della distanza di Levenshtein sono: cancellazione, sostituzione e inserimento.

Costi operazioni elementari

In base al costo che viene attribuito ad ognuna delle operazioni elementari si ottengono misure diverse di similarità tra stringhe. La distanza di Levenshtein associa i seguenti costi:

- cancellazione di un carattere → costo: 1
- sostituzione di un carattere con un altro → costo: 2
- inserimento di un carattere → costo: 1

Inoltre si assume che l'operazione di sostituzione di un carattere con se stesso abbia costo 0.

2.5.1 Calcolo della distanza di Levenshtein

In questa sotto sezione viene illustrato il calcolo della distanza di Levenshtein con un metodo basato sulla programmazione dinamica.

La difficoltà relativa al calcolo della distanza di Levenshtein può essere vista come un problema di ricerca nel quale si individua la più corta sequenza di operazioni per trasformare una stringa in un'altra.
Lo spazio di ricerca che comprende tutte le possibili sequenze di operazioni è estremamente vasto, però si nota che molti percorsi di modifica distinti finiranno nello stesso stato o stringa. Quindi invece di calcolare tutti questi percorsi si può semplicemente ricordare il percorso più breve verso uno stato o stringa e utilizzarlo ogni volta che lo si

incontra.
La tecnica della programmazione dinamica [BM14], introdotta da Bellman nel 1957, viene applicata per risolvere il problema del calcolo della distanza di Levenshtein.

Descrizione algoritmo

Si considerino:

- due stringhe X(source) e Y(target);
- due interi n e m ottenuti nel seguente modo: $|X| = n$ e $|Y| = m$;
- una matrice D con n+1 righe e m+1 colonne;
- D[i,j] rappresenta la distanza di Levenshtein tra le stringhe:
  - X[1,i] cioè i primi i caratteri di X;
  - Y[1,j] cioè i primi j caratteri di Y.

  Quindi la distanza di Levenshtein totale tra X e Y si troverà in D[n,m]

Per applicare la programmazione dinamica si deve trovare una formulazione ricorsiva del problema che abbia le proprietà di sotto struttura ottima. La seguente funzione definisce il calcolo di D[i,j]:

$$D[i,j] = min \begin{cases} D[i-1,j] + \text{costo\_canc(source[i])} \\ D[i,j-1] + \text{costo\_ins(target[j])} \\ D[i-1,j-1] + \text{costo\_sost(source[i],target[j])} \end{cases}$$

Con i costi delle operazioni elementari, descritti in precedenza, la funzione diviene:

$$D[i,j] = min \begin{cases} D[i-1,j] + 1 \\ D[i,j-1] + 1 \\ D[i-1,j-1] + \begin{cases} 2 \text{ se source[i]} \neq \text{target[j]} \\ 0 \text{ se source[i]} = \text{target[j]} \end{cases} \end{cases}$$

3. TLSH

In questo capitolo viene illustrato in modo approfondito il funzionamento dell'algoritmo TLSH [OCC13] poiché su di essa si basa l'implementazione del sistema del progetto di tesi.
La funzione TLSH (TrendMicro Locality Sensitive Hashing) fu ideata dalla multinazionale Trend Micro Inc nel 2013 con lo scopo di trovare uno schema di Locality Sensitive Hashing per la ricerca di file simili.
La funzione TLSH utilizza le tecnologie che sono risultate migliori in termini di efficienza e applicabilità al problema della ricerca degli oggetti simili, nell'analisi svolta nel capitolo precedente:

- Locality Sensitive Hashing (LSH) (vedi sotto sezione 2.3.2)
- Context Triggered Piecewise Hashing (CTPH) (vedi sotto sezione 2.2.3)
- Feature extraction (vedi sezione 2.4)

3.1 Costruzione del digest TLSH

In questa sezione vengono descritte nel dettaglio tutte le fasi del processo di creazione del digest TLSH.

Si considerino i byte della stringa in input, avente lunghezza *len*, nel seguente modo:

Byte[0], Byte[1], ... , Byte[len-1]

Il digest TLSH è una stringa formata da 35 byte ottenuta attraverso il seguente processo:

1. Elaborazione dei byte della stringa con sliding window (3.1.1).
2. Calcolo dei quartile points (3.1.2).

3. Calcolo dell'header del digest (3.1.3).

4. Calcolo del body del digest (3.1.4).

5. Costruzione del digest completo (3.1.5).

Le fasi sopra elencate vengono descritte nel dettaglio nelle seguenti sotto sezioni.

3.1.1 Elaborazione dei byte della stringa con sliding window

I byte della stringa vengono processati attraverso una sliding window di dimensione 5 per popolare un array di contatori di bucket mediante la seguente procedura:

```
1  //inizializzazione di tutti i bucket a 0
2  //ew: fine della sliding window
3  //sw: inizio della sliding window
4  for ew=4 to len-1{
5    sw=ew-4
6    for tri=1 to 6{
7      (c1,c2,c3)=Triplet(tri,Byte[sw...ew])
8      bi=b_mapping(c1,c2,c3)
9      bucket[bi]++
10   }
11 }
```

Listing 3.1: Elaborazione dei byte in TLSH

La proprietà fondamentale delle funzioni di ricerca dei Nearest Neighbors, riguardante la creazione di digest simili per oggetti simili, viene rispettata dalla funzione TLSH grazie all'utilizzo della una sliding window per l'elaborazione dei byte. Piccole modifiche sui file corrispondono a piccole modifiche del valore dell'hash calcolato da TLSH.

Descrizione della procedura

- Vengono utilizzati 256 bucket perché un carattere viene rappresentato da un byte e in tale byte possono essere rappresentati interi compresi tra 0 e 255.
  I contatori dei bucket vengono inizializzati a 0.

- Viene utilizzata una sliding window di dimensione 5 per estrarre le triplette. La dimensione 5 della sliding window venne utilizzata precedentemente nel Nilsimsa hash [De +04] e ne venne dimostrata l'efficacia.

- La variabile *tri* varia tra i valori 1 e 6, perché non vengono considerate alcune triplette per evitare ripetizioni. Dati i 5 caratteri considerati dalla sliding window (A,B,C,D,E) esistono 10 possibili triplette:

  1. A B C
  2. A B D
  3. A B E
  4. A C D
  5. A C E
  6. A D E
  7. B C D
  8. B C E
  9. B D E
  10. C D E

  Le triplette dalla 7 alla 10 vengono escluse perché vengono elaborate da un successivo spostamento della finestra scorrevole; di conseguenza, si ottengono in totale 6 triplette.

- Al termine dell'esecuzione della funzione *Triplet* la variabile (c1,c2,c3) contiene la tripletta su cui viene eseguito il mapping nei bucket, con la funzione *b_mapping*.
  La funzione *b_mapping* implementa la funzione di mapping chiamata Pearson hash [Pea90]. In TLSH tale funzione viene utilizzata per il mapping nei bucket, infatti il valore restituito da *b_mapping* viene utilizzato come indice per selezionare il bucket nell'array bucket.

- La parte di incremento dei bucket, $bucket[bi]++$, viene eseguita solo se la sliding window contiene almeno 4 caratteri.

Pearson hash

Il Pearson Hash [Pea90] prende in input una stringa w composta da un numero n di byte di caratteri $(c_1, c_2, ..., c_n)$ e restituisce in output un valore compreso tra 0 e 255.
Il calcolo del valore in output avviene utilizzando una tabella ausiliaria T nel seguente modo:

```
1  //h: valore in output
2  //T: tabella ausiliaria
3  h[0]=0
4  for i in 1,...,n{
5    h[i]=T[h[i-1] XOR C[i]]
6  }
7  return h[n]
```

Listing 3.2: Pearson Hash

Come evidenziato dal codice, il Pearson hash può restituire solo i valori che appaiono in T; di conseguenza, T deve essere una permutazione dei valori tra 0 e 255.
Il Pearson hash presenta le seguenti caratteristiche:

- L'elaborazione di ciascun carattere aggiuntivo richiede solo un'operazione di OR esclusivo e una lettura da memoria.
- Non è necessario conoscere la lunghezza della stringa all'inizio calcolo. Questa proprietà è molto utile quando la fine della stringa di testo è indicata da un carattere speciale anziché da una variabile di lunghezza memorizzata separatamente.
- Piccole modifiche ai dati di input comportano grandi modifiche, apparentemente casuali, al valore hash.
  In TLSH questa proprietà del Pearson Hash comporta l'incremento di contatori molto lontani nel caso in cui la tripletta (c1,c2,c3) contenga caratteri simili a quelli di una tripletta precedentemente considerata.
- Dato un valore hash è computazionalmente impossibile trovare il valore di input che lo ha generato.

3.1.2 Calcolo dei quartile points

In statistica, i *quartile points* sono punti che suddividono un set di dati in quattro parti; in base ai valori assunti da tali punti è possibile effettuare considerazioni importanti sul dataset.
In TLSH il set di dati considerato dal calcolo dei quartile points è rappresentato dai contatori dei bucket determinati durante l'elaborazione dei byte della stringa con sliding window.
I valori dei*quartile points* (q1,q2,q3) vengono valutati nel seguente modo:

- 75% dei contatori di bucket $\geqslant$ q1 (quartile inferiore);
- 50% dei contatori di bucket $\geqslant$ q2 (quartile centrale);
- 25% dei contatori di bucket $\geqslant$ q3 (quartile superiore).

3.1.3 Costruzione dell'header del digest

L'header del digest contiene informazioni riguardanti la stringa su cui si sta calcolando il digest TLSH.
L'header è composto dai primi tre byte del digest, i quali contengono i seguenti valori:

- 1° byte: checksum modulo 256 della stringa.
- 2° byte: valore del logaritmo in base 2 della lunghezza in byte della stringa modulo 256:

$$log_2(len\ MOD\ 256)$$

- 3° byte: due quantità di 16 bit derivate dai quartile points nel seguente modo:
  - $q1_{ratio} = (\frac{q1 \cdot 100}{q3})\ MOD\ 16$
  - $q2_{ratio} = (\frac{q2 \cdot 100}{q3})\ MOD\ 16$

  Le formule per i q_{ratio} sono state determinate sperimentalmente.

Motivazione checksum

Il checksum nel primo byte dell'header viene utilizzato per evitare falsi positivi e per accertarsi che i digest che si trovano in bucket vicini della tabella hash abbiano una distanza maggiore di 0.
Nell'implementazione della funzione TLSH l'utilizzo o meno del checksum è configurabile.

Motivazione secondo byte dell'header

Il valore del secondo byte dell'header viene utilizzato per poter identificare come simili stringhe con medesima struttura ma di dimensioni molto diverse che altrimenti non verrebbero considerate tali.
Nell'implementazione della funzione TLSH l'utilizzo o meno del secondo byte dell'header è configurabile.

3.1.4 Costruzione del body del digest

Il body è contenuto negli ultimi 32 byte del digest.
Il calcolo del valore del body viene eseguito confrontando tutte le posizioni dell'array dei contatori dei bucket con i quartile points e in base al risultato di tale confronto viene restituita una coppia di bit. Infine le coppie di bit vengono concatenate per ottenere il body del digest.
Le coppie di bit vengono restituite in base al seguente meccanismo:

```
1  for bi=0 to 255{
2    if(bucket[bi]<=q1){
3      Emit(00)
4    }
5    else if(bucket[bi]<=q2){
6      Emit(01)
7    }
8    else if(bucket[bi]<=q3){
9      Emit(10)
10   }
11   else{
12     Emit(11)
```

```
13      }
14  }
```

Listing 3.3: Costruzione del body del digest TLSH

3.1.5 Costruzione del digest completo

Per creare il digest completo si concatenano le seguenti parti:

1. Rappresentazione esadecimale dell'header del digest;
2. Rappresentazione esadecimale del body del digest.

Il digest così generato viene usato nel calcolo della differenza tra i digest nella ricerca della similarità.

3.2 Scoring delle distanze tra i digest TLSH

In questa sezione viene descritto nel dettaglio il calcolo della distanza tra due digest di TLSH.

A differenza delle funzioni hash ssdeep e sdhash, TLSH calcola il distance score tra due digest. Il distance score può assumere valori positivi con il seguente significato:

- Distance score = 0: i file sono identici;
- Distance score > 0: i file sono diversi e maggiore è tale valore allora maggiori sono le differenze.

Il distance score non ha un valore massimo.
Nell'implementazione del sistema del progetto di tesi si considerano simili i file aventi distance score < 100; tale valore è stato ottenuto sperimentalmente.

Il distance score tra due digest tx e ty è definito nel seguente modo:

$$\text{distance\_score(tx,ty)} = \text{distance\_header(tx,ty)} + \text{distance\_body(tx,ty)}$$

dove tx e ty sono due valori esadecimali dalle quali possiamo estrarre: checksum, lvalue, $q1_{ratio}$ e $q2_{ratio}$.

3.2.1 Calcolo del distance header tra due digest

In questa parte dell'elaborato viene descritto il calcolo eseguito dalla funzione distance_header(tx,ty) di TLSH.

Definizione funzione mod_diff

Il calcolo del distance score utilizza la funzione mod_diff per valutare la distanza tra le varie parti dell'header del digest. La funzione mod_diff(x,y,R) calcola il numero minimo di passi tra x e y in una coda circolare di lunghezza R.
Esempio: mod_diff(15,3,16)=4 perché $15 \longrightarrow 0 \longrightarrow 1 \longrightarrow 2 \longrightarrow 3$

Implementazione calcolo distance header

```
1  function distance_header(tx,ty){
2    int diff=0;
3
4    //calcolo differenza tra il byte riguardanti la lunghezza
5    //della stringa
6    ldiff=mod_diff(tx.lvalue, ty.lvalue, 256);
7    if(ldiff<=1){
8      diff=diff+ldiff ;
9    }
10   else{
11     diff=diff+ldiff*12;
12   }
13
14   //calcolo differenza tra i valori dei q_ratio
15   q1diff=mod_diff(tx.q1ratio, ty.q1ratio, 16);
16   if(q1diff<=1){
17     diff=diff+q1diff;
18   }else{
19     diff=diff+(q1diff-1)*12;
```

```
20    }
21    q2diff=mod_diff(tx.q2ratio, ty.q2ratio, 16);
22    if(q1diff<=1){
23      diff=diff+q2diff;
24    }else{
25      diff=diff+(q2diff-1)*12;
26    }
27
28    //calcolo differenza tra i checksum
29    if(tx.checksum <> ty.checksum){
30      diff=diff+1
31    }
32    return diff;
33  }
```

Listing 3.4: Calcolo della distanza tra gli header dei digest di TLSH

Nella funzione distance_header(tx,ty) il range passato alla funzione mod_diff è 16 per i q_{ratio}, questo perché la sua dimensione è di 16 bit. Al contrario, il range passato alla funzione mod_diff per i valori lenght dell'header è 256 perché tali valori vengono memorizzati nel secondo byte del digest e quindi utilizzano 256 bit.
Il parametro 12 utilizzato nella funzione distance_header(tx,ty) è stato calcolato sperimentalmente.

3.2.2 Calcolo del distance body tra due digest

In questa parte dell'elaborato viene descritto il calcolo eseguito dalla funzione distance(tx,ty) di TLSH.

La funzione distance_body(tx,ty) calcola un'approssimazione della distanza di Hamming. La differenza con la distanza di Hamming si ha con l'aggiunta del valore 6 alla distanza quando i contatori dei bucket in tx e ty sono posizionati rispettivamente uno nel quartile inferiore e l'altro nel quartile superiore.

La distanza tra i body di due digest tx e ty viene calcolato nel seguente modo:

```
function distance_body(tx,ty){
  int diff=0;
  for i=0 to 64{
    x1=tx.hex[i+5]/4;
    x2=tx.hex[i+5]%4;
    y1=ty.hex[i+5]/4;
    y2=ty.hex[i+5]%4;
    d1=abs(x1-y1);
    d2=abs(x2-y2);

    if(d1==3){
      diff=diff+6;
    }
    else{
      diff=diff+d1;
    }
    if(d2==3){
      diff=diff+6;
    }
    else{
      diff=diff+d2;
    }
  }
  return diff;
}
```

Listing 3.5: Calcolo della distanza tra i body dei digest di TLSH

Per quanto concerne il parametro 6, i calcoli effettuati per il suo ottenimento non risultano essere interessanti ai fini dell'elaborato di tesi.

3.3 Test di confronto tra funzioni LSH

La scelta di usare TLSH per implementare il sistema software del progetto di tesi è motivata dagli ottimi risultati ottenuti da tale funzione nei test descritti nell'articolo [OFC14] e riassunti in questa sezione.

3.3.1 Funzioni per il confronto

In questa sotto sezione vengono descritte due funzioni che vengono confrontate con TLSH nell'articolo [OFC14].
La descrizione di queste due funzioni è volutamente poco approfondita e ha come unico scopo quello di illustrare le differenza di funzionamento delle due funzioni rispetto a TLSH.

ssdeep

Ssdeep utilizza tre fasi per la creazione del digest:

1. Divisione del documento in segmenti distinti mediante l'utilizzo di un *rolling hash* [Rou10], anche detto hash ricorsivo.

2. Produce un valore di 6 bit per ogni segmento, rappresentante l'hash del segmento.

3. Concatena gli hash di segmento in base64 per creare il digest.

Calcolo della distanza in ssdeep:
Ssdeep assegna un punteggio di similarità nell'intervallo 0-100 calcolando la distanza chiamata *edit distance* o *distanza di Levenshtein* (vedi 2.5), questa misura di distanza viene calcolata contando il numero minimo di operazioni richieste per trasformare un digest nell'altro.

sdhash

Sdhash utilizza tre fasi per la creazione del digest:

1. Identifica sequenze di 64 byte che hanno caratteristiche che sono statisticamente improbabili (SIF: Statistically Improbable Features), cioè seleziona le caratteristiche che hanno una granularità adatta ad identificare gli elementi dello spazio di oggetti che stiamo analizzando.

2. Calcola il digest delle sequenze precedentemente selezionate e li inserisce nei *Bloom Filter*.

I Bloom Filter sono vettori di bit progettati per rispondere al problema decisionale di appartenenza di un elemento ad un insieme in modo rapido ed efficiente in memoria.

3. Codifica dei filtri Bloom per formare il digest.

Calcolo della distanza in sdhash:
Sdhash assegna un punteggio di similarità nell'intervallo 0-100 calcolando la distanza chiamata *normalized entropy* tra due digest.

3.3.2 Test di confronto

Nel test vengono confrontate le funzioni LSH: ssdeep, sdhash e TLSH. Queste tre funzioni in particolare vengono scelte poiché largamente utilizzate ed inoltre online sono disponibili le loro implementazioni come codice open source [TLSH] [ssdeep].
Il confronto si basa su 3 tipi di test volti ad analizzare i risultati delle funzioni con diversi tipi di file.

Ai fini dello studio svolto in questo progetto di tesi si è interessati unicamente al test relativo ai file di testo e pagine web sui quali sono state apportate modifiche casuali per simulare un ambiente contraddittorio. Da tale test si sono ottenuti i seguenti risultati:

- Le modifiche casuali hanno avuto successo nell'ottenere risultati inconsistenti dalle sdhash e ssdeep con file di dimensione inferiore a 40.000 byte.
- TLSH con dimensioni di file comprese tra 20.000 e 40.000 byte risulta ancora in grado di identificare con successo i file correlati.

In conclusione, TLSH appare significativamente più robusto nel riconoscimento di modifiche casuali e alle manipolazioni contraddittorie rispetto a ssdeep e sdhash.

4. Similarity Hashing Engine di TLSH

Questo capitolo ha l'obiettivo di descrivere la struttura dell'ambiente del progetto di tesi e di analizzare le soluzioni apportate dall'utilizzo di TLSH nella ricerca dei Nearest Neighbors.

L'indagine affrontata all'interno dell'elaborato di tesi è costituita dal problema formale della ricerca dei Nearest Neighbors applicato all'esigenza di suggerire testi simili a quelli in corso di scrittura per un autore di documentazione tecnica aziendale. Al fine di velocizzare la stesura e la traduzione dei documenti, l'azienda ha bisogno di un ambiente in grado di effettuare una ricerca veloce ed efficace di stringhe di testo simili a quella digitata dal dipendente dell'azienda. Tale ricerca va effettuata sulla collezione di documenti di quest'ultima contenente i testi corretti e validati.
L'ambiente ideato per rispondere alle esigenze dell'azienda utilizza la funzione TLSH per la ricerca di similarità.

4.1 Casi d'uso di LSH e Text Mining

In questa sezione vengono descritti alcuni campi di applicazione delle funzioni basate su Locality Sensitive Hashing e alcuni impieghi del Text Mining.

4.1.1 Pattern recognition

Il termine pattern recognition viene utilizzato per descrivere le tecniche finalizzate al riconoscimento automatico di schemi e regolarità nei dati.
Il Locality Sensitive Hashing e gli algoritmi basati su tale tecnica sono utilizzati principalmente nell'ambito Optical Character Recognition (OCR) che si occupa della conversione di immagini di testo digitato, scritto a mano o stampato in caratteri del formato ASCII

o Unicode. Un esempio di tale utilizzo è presente nell'articolo: Robust Recognition of Documents by Fusing Results of Word Clusters [Ras+09].

4.1.2 Computer vision

Computer Vision è un sotto campo dell'intelligenza artificiale il cui obiettivo è costruire un computer che replichi l'intelligenza visiva del cervello umano.
I predominanti campi di applicazione della Computer vision sono:

- Controllo veicoli autonomi: warning-system nelle automobili, sistemi per l'atterraggio automatico degli aerei o sistemi per la guida automatica di autovetture.
- Ambito medico per l'estrazione di informazioni dalle immagini con l'intento di effettuare una diagnosi di un paziente. Tipicamente l'immagine è acquisita attraverso microscopia, raggi X, angiografia e tomografia.

In diversi casi le tecniche di LSH sono state applicate a tali problemi. Inoltre la funzione TLSH è in grado di rilevare similarità tra immagini in modo molto più efficace rispetto a sdhash e ssdeep. [OFC14]

4.1.3 Similarità tra documenti

Un'importante classe di problemi a cui la tecnica di LSH può essere applicata è rappresentata dalla ricerca di documenti testualmente simili all'interno di una collezione di testi molto vasta. [De +04]
Tale ricerca viene applicata a diversi campi, tra i quali: rilevazione del plagio, rilevazione delle pagine mirror.

4.1.4 Rilevazione dei malware

Le tecniche di LSH e più in generale tutte le tecniche utilizzate per la ricerca dei Nearest Neighbors sono state utilizzate all'interno di software anti-malware [Bay+09] [Rou11].
TLSH è stato ideato dalla TrendMicro per il whitelisting di file [trendmicroblog], cioè per determinare se un file è sicuro o meno in base alla sua somiglianza con file noti e

legittimi. In precedenza, nella maggior parte dei casi, per identificare applicazioni sicure venivano utilizzate le seguenti politiche:

- File presenti nella whitelist;
- File con un certificato firmato da un fornitore di software considerato affidabile.

TLSH aggiunge una nuova politica alle due precedenti: i file simili a quelli nella whitelist e firmati dallo stesso fornitore di software.
I gruppi di file ottenuti con la politica basata su TLSH possono contenere malware poiché componenti dannosi possono essere stati inseriti come applicazioni legittime. Per evitare questi casi, i test di somiglianza con TLSH possono essere combinati con l'analisi di ulteriori informazioni sul file, come i firmatari di certificati attendibili.

4.1.5 Applicazione biomedica

Numerosi studi [BioNLP] in campo biomedico sono stati effettuati sull'applicazione del Text Mining ad approcci computazionali sull'aggancio e l'interazione tra proteine e associazioni proteina-malattia.
Inoltre l'estrazione di informazione dai testi, applicata alle numerose documentazioni cliniche dei pazienti, possono facilitare l'indicizzazione di eventi clinici in specifici insiemi di dati testuali e l'individuazione di eventuali pattern ricorrenti con lo scopo di agevolare le diagnosi su altri pazienti.
Applicazioni di text mining disponibili online in campo biomedico sono PubGene e GoPubMed, motori di ricerca che combinano le tecniche di data mining a testi biomedici con la visualizzazione di rete.

4.1.6 Sentiment analysis

Per sentiment analysis (noto anche come opinion mining o emotion AI) si intende l'uso dell'elaborazione del linguaggio naturale, analisi del testo con tecniche di text mining e linguistica computazionale per identificare, estrarre, quantificare e studiare sistematicamente stati affettivi e informazioni soggettive da testi digitati dagli utenti.
La sentiment analysis è largamente applicata a testi relativi alle recensioni degli utenti,

risposte ai sondaggi e materiali sanitari per applicazioni che vanno dal marketing alla medicina clinica.

4.2 Funzionalità e utilità del sistema

In questa sezione vengono illustrate le funzionalità del sistema e viene descritto in che modo esse risultano utili per la creazione dell'ambiente richiesto dal progetto di tesi.

4.2.1 Funzionalità del sistema

Il sistema è formato da due parti: ricerca della similarità con la funzione TLSH e API (Application Programming Interface) per il confronto tra varie tecniche di LSH.

La parte del sistema riguardante la ricerca della similarità con TLSH prende in input una stringa formattata in html esteso, secondo le specifiche del progetto. La stringa in input viene suddivisa in sezioni, paragrafi e frasi, secondo uno specifico schema legato all'html esteso. Infine viene eseguita la ricerca di similarità con la funzione TLSH.
Al termine della ricerca di similarità il sistema restituisce due array:

- array delle stringhe del data set che risultano essere simili ad una specifica parte dell'input;
- array di sezioni, paragrafi e frasi in cui l'input viene suddiviso.

Il sistema mette a disposizione un API che viene utilizzata in fase di test per il confronto dell'implementazione dell'ambiente TLSH con le implementazioni di altre tecniche di LSH, svolte da laureandi magistrali, al fine di comprendere quale sia l'approccio più conveniente per implementare al meglio l'ambiente.

4.2.2 Utilità del sistema

Il progetto ha lo scopo di gestire grandi quantità di documenti che devono rispettare una determinata struttura. Al fine di facilitare la stesura di tali documenti è stato ideato un html esteso in base alle regole di struttura e organizzazione del contenuto richieste

dall'azienda. Inoltre al fine di velocizzare la stesura e la traduzione di questi documenti è necessario aggiungere un meccanismo in grado di ricercare porzioni simili di testo all'interno dell'insieme dei documenti dell'azienda. Per fare in modo che tale ricerca sia più funzionale e specifica si effettua tale analisi in relazione al tipo della porzione di documento considerata. I valori assunti da tale tipo sono: sezione, paragrafo, frase e trigramma.
Il progetto di tesi ha come scopo quello di confrontare il funzionamento di TLSH con quello delle funzioni Minhash e Simhash al fine di creare l'ambiente più adatto possibile alle richieste dell'azienda.

4.3 Problemi preesistenti e soluzioni di TLSH

L'obiettivo della seguente sezione è quello di descrivere in che maniera TLSH risolve i problemi delle tecnologie preesistenti usando tecniche più adeguate alla ricerca di documenti di testo simili.
Le tecniche citate in questa sezione sono state illustrate nei particolari nelle sezioni 2.2, 2.3 e 2.4.

4.3.1 Problemi di creazione del digest

Nella sezione 2.2 sono state descritte tre diverse tecniche per la creazione di digest di oggetti.
La prima tecnica, riguardante le funzioni hash crittografiche, calcola un unico digest per tutto l'oggetto producendo digest diversi per oggetti simili.
La seconda tecnica denominata Block-Based Hashing, cerca di restringere la porzione di digest alterata in seguito alla modifica di un oggetto. A tale fine i Block-Based Algorithm suddividono l'oggetto in parti fisse e calcolano il digest come la concatenazione dei valori hash di ogni blocco. Tuttavia, anche questo metodo non risulta essere applicabile alla ricerca degli oggetti simili a causa del problema di allineamento in caso di inserimento o rimozione di caratteri all'interno dell'oggetto considerato.
Infine l'ultima tecnica, chiamata CTPH, cerca di eliminare il problema di allineamento suddividendo in modo pseudocasuale l'oggetto in input. Quest'ultima risulta essere la

tecnica più adeguata nella ricerca dei Nearest Neighbors ed è la tecnica utilizzata da TLSH per la creazione del digest.

4.3.2 Problema di efficienza

La tecnica del Locality Sensitive Hashing, su cui si basa TLSH, è una soluzione migliore rispetto a quella degli algoritmi di Approximate Matching perché non richiede una ricerca lineare sulla collezione di documenti.
In LSH si considerano solo le coppie che probabilmente sono simili senza indagare su ogni coppia possibile; di conseguenza grazie a tale tecnica si ha un notevole abbassamento del costo computazionale (vedi sotto sezione: 2.3.2).

4.3.3 Problema dei falsi positivi

Le tecniche per il datafingerprinting, introdotte da Rabin nel 1981, si basano sulla selezione casuale delle caratteristiche. Tali tecniche applicate alla ricerca degli oggetti simili soffrono di problemi legati all'alto numero di falsi positivi. Al contrario, la tecnica di Feature Extraction utilizzata da TLSH chiamata Non-Rabin Fingerprinting non presenta questi limiti.

4.4 Interfaccia grafica e struttura del sistema

In questa sezione viene illustrato il mockup del sistema e vengono descritti esempi di utilizzo dei servizi del sistema.

4.4.1 Mockup

In questa sotto sezione viene illustrato il mockup dell'ambiente disponibile all'indirizzo web http://site181901.tw.cs.unibo.it.
La progettazione grafica dell'ambiente è stata eseguita mediante il software Balsamiq [balsamiq] al fine di renderne più semplice la trasposizione nei layout html; le immagini generate con tale strumento vengono riportate di seguito.

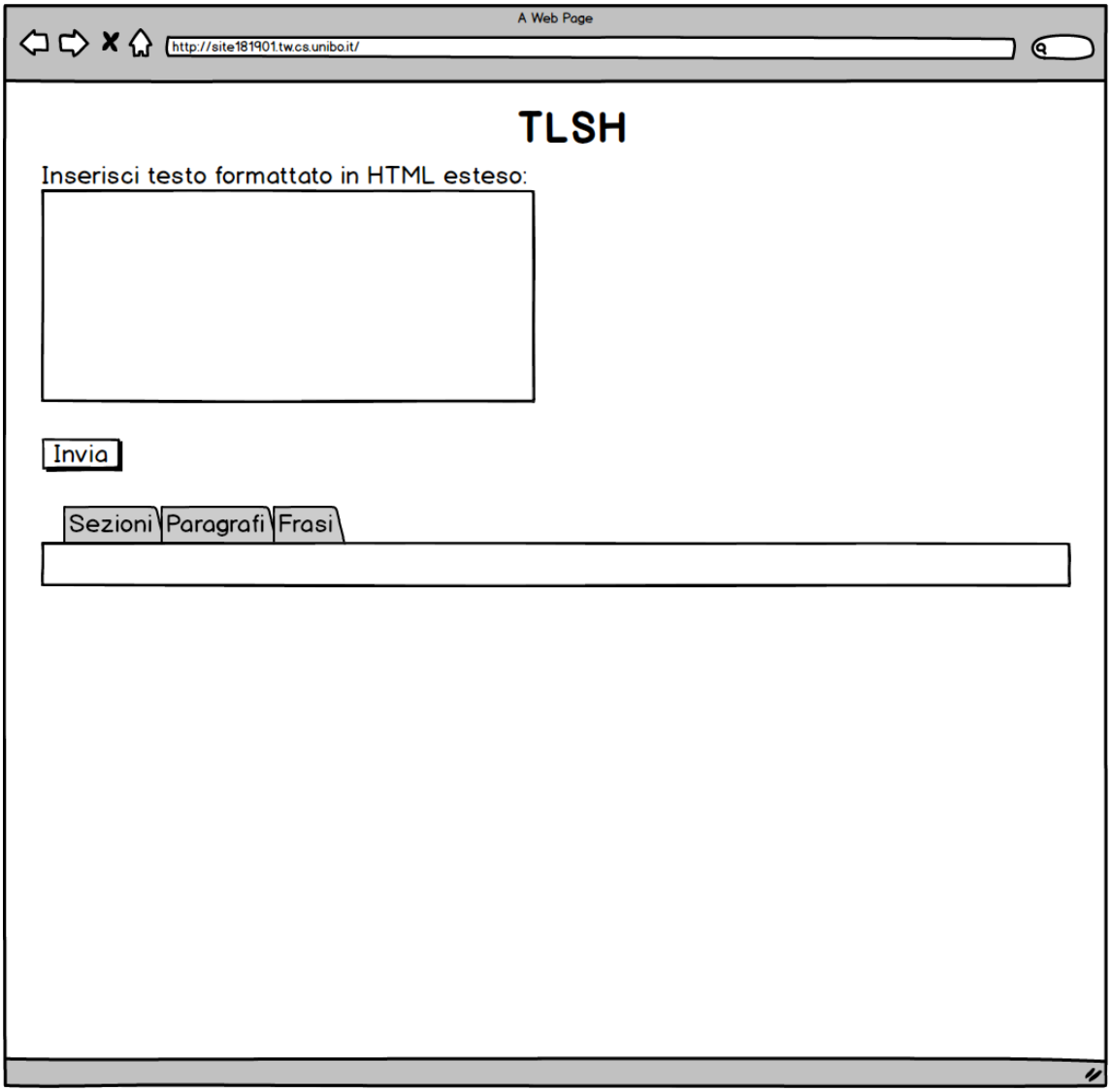

Figura 4.1: Mockup della pagina web prima dell'invio del testo

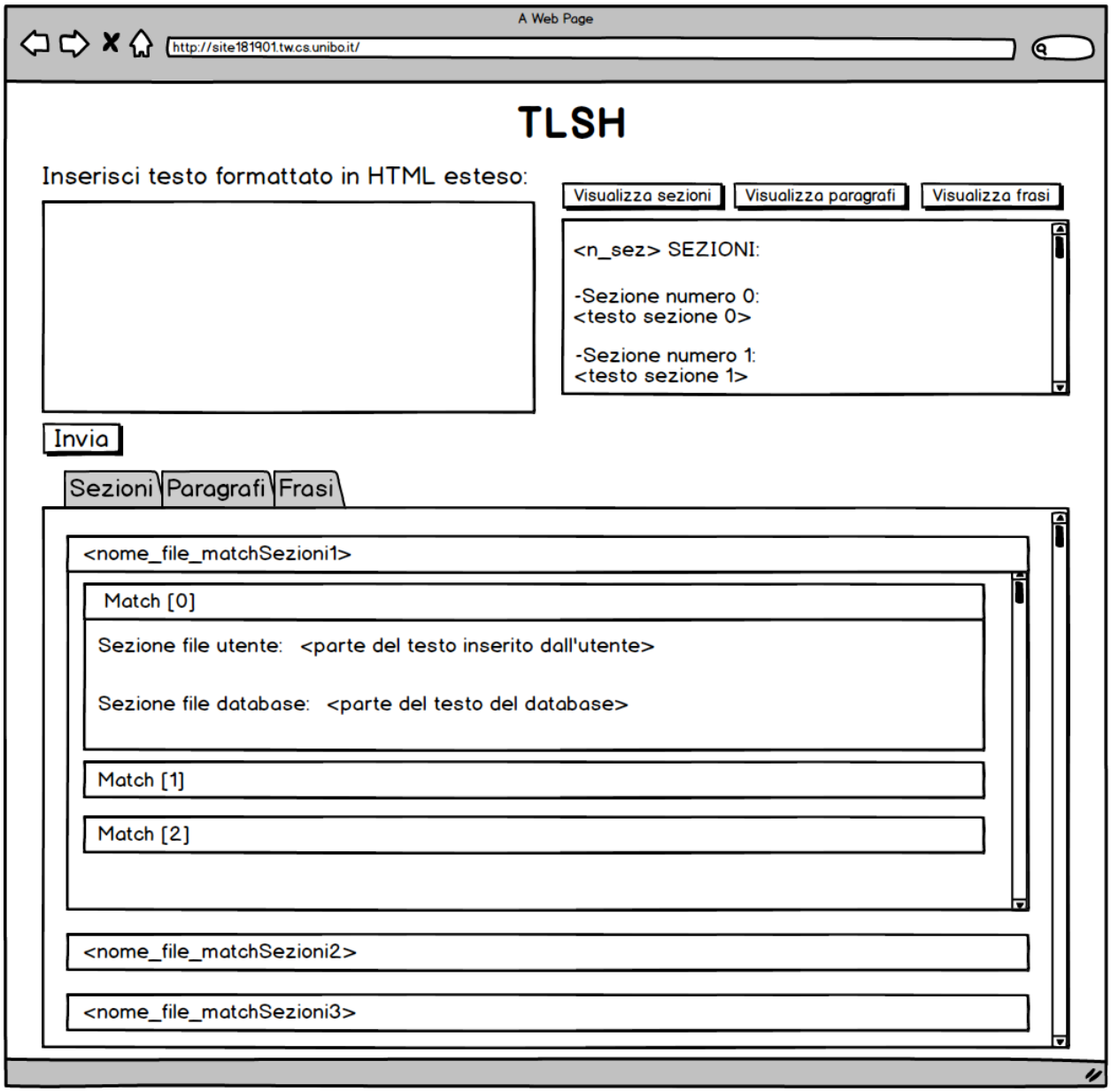

Figura 4.2: Mockup della pagina web dopo dell'invio del testo

4.4.2 Implementazione interfaccia grafica

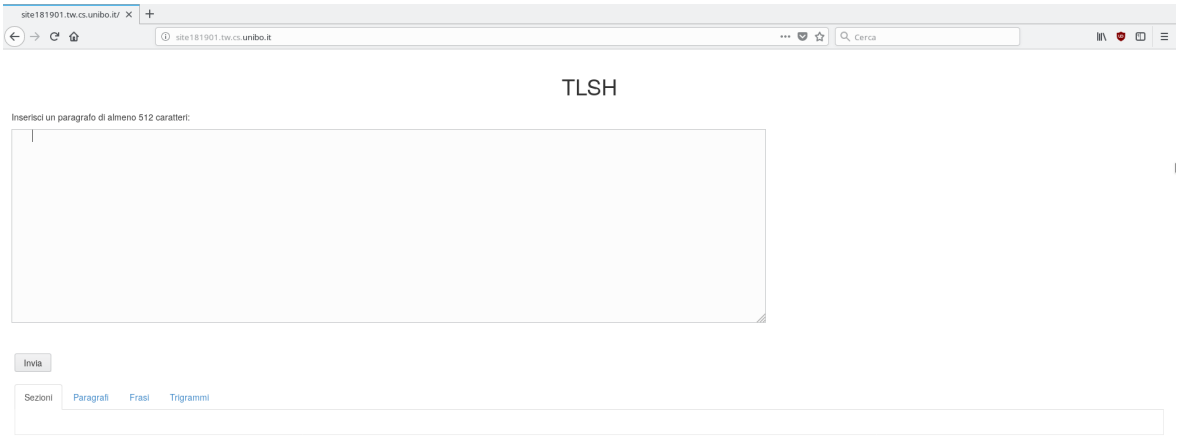

Figura 4.3: Interfaccia grafica della pagina web prima dell'invio del testo

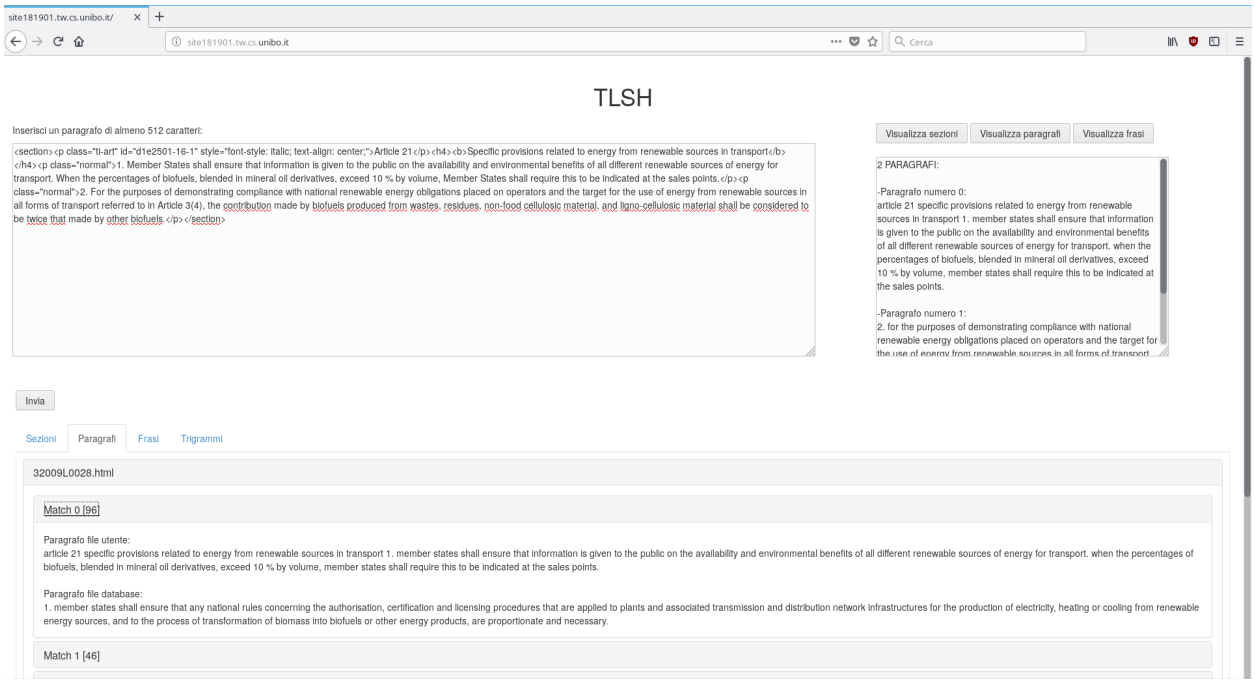

Figura 4.4: Interfaccia grafica della pagina web dopo dell'invio del testo

4.4.3 Esempi di utilizzo dei servizi del sistema

I servizi messi a disposizione dall'ambiente sono due: test TLSH e API per il confronto con le funzioni LSH. Per il primo servizio vengono descritti i passi che un eventuale utente deve eseguire per utilizzare tale parte dell'ambiente. Per il secondo servizio viene descritto il funzionamento generale dell'API.

Utilizzo test TLSH

L'ambiente di test di TLSH prende in input un testo formattato secondo l'html esteso e restituisce la lista delle sezioni, dei paragrafi e delle frasi simili alle porzioni di testo passate come parametro. Questo ambiente è disponibile all'indirizzo web:
http://site181901.tw.cs.unibo.it/.
L'utente inserisce nella text area il testo in html esteso e clicca sul bottone di invio sottostante. Al termine dell'elaborazione dei dati e del calcolo dei risultati le diverse liste vengono rese visibili all'utente nei relativi tab posti sotto il bottone di invio. Inoltre, viene mostrata una text area e tre diversi bottoni per la visualizzazione delle diverse porzioni di testo, suddivise per tipo.

Utilizzo API per confronto con le funzioni LSH

L'API viene utilizzata per testare l'applicazione e successivamente confrontarla con altre funzioni LSH. Essa richiede una stringa di testo in input e restituisce la lista delle porzioni di testo simili in base al campo type della richiesta all'API.
I campi della richiesta e della risposta dell'API vengono descritti in 5.2.

5. Implementazione del sistema

In questo capitolo viene descritta la struttura del progetto e l'implementazione di ogni sua parte.
Come descritto precedentemente, il sistema è formato da due parti distinte: l'ambiente di test della ricerca di similarità con la funzione TLSH e il servizio riguardante l'API per il confronto tra funzioni LSH. Per ognuno di questi servizi vengono descritte nel dettaglio le principali scelte implementative.

5.1 Ambiente per il test di TLSH

L'ambiente per il test di TLSH prende in input una stringa di testo formattata nell'HTML esteso e restituisce in output due array con i seguenti contenuti: stringhe del data set che risultano essere simili ad una specifica parte dell'input e lista di sezioni, paragrafi e frasi in cui l'input viene suddiviso.

5.1.1 HTML esteso

In questa sottosezione vengono illustrate le regole utilizzate per la suddivisione del testo in input in sezioni, paragrafi e frasi. Questa suddivisione utilizza regole basate sull'HTML esteso ideato per i documenti dell'azienda al fine di unificare la struttura dei documenti, rendere più rapida la verifica della conformità e facilitarne la conversione in diversi formati in base alle necessità.
L'HTML esteso ideato per tali scopi viene rispettato da tutti i documenti dell'azienda. Nel sistema basato su TLSH viene sfruttata la struttura fornita da tale HTML esteso per suddividere documenti o porzioni di essi in sezioni, paragrafi e frasi. Tale suddivisione avviene secondo le seguenti regole:

- *Sezione*: sequenza di testo identificata da tag **<section>**.

- *Paragrafo*: sequenza di testo che termina con **.</p>** .
- *Frase*: sequenza di testo delimitata da punti, punti e virgola e due punti.
  La regex utilizzata nel codice richiede che prima dei caratteri di punteggiatura usati per la suddivisione ci siano almeno 3 caratteri; tale scelta è stata fatta per evitare di considerare come frasi anche gli elenchi numerati.

5.1.2 Caricamento iniziale del data set

L'ambiente di TLSH deve considerare un grande numero di documenti in cui cercare stringhe simili. Al fine di evitare il caricamento ripetuto dello stesso data set e calcolare più volte gli hash, è stato creato un json contenente tutte le informazioni necessarie.
Il caricamento iniziale del data set è avvenuto nel seguente modo:

1. Caricamento dei file presenti al seguente indirizzo web:
   http:// site181928.tw.cs.unibo.it/json_train/.
   Tali file json hanno la seguente struttura:

```
{
  data:{
    <nome_file1>:<array sezioni/paragrafi/frasi 1>,
    <nome_file2>:<array sezioni/paragrafi/frasi 2>,
    ...
  }
}
```

Listing 5.1: Struttura dei file di train

2. Creazione dell'array fileDB contenente oggetti di tipo InfoFile.
   In questa fase viene creato un oggetto di tipo InfoFile per ogni file presente nei file json caricati al punto precedente.
   L'array fileDB viene popolato da oggetti InfoFile creati nel seguente modo:
   - Creazione dell'oggetto InfoFile passando come parametri al costruttore le seguenti informazioni:
     - nome del file

- array delle sezioni del file considerato recuperato dal file total_section.json
- array dei paragrafi del file considerato recuperato dal file total_paragraph.json
- array delle frasi del file considerato recuperato dal file total_phrase.json.

- Calcolo dei valori hash TLSH di tutte le sezioni.
- Calcolo dei valori hash TLSH di tutti i paragrafi.
- Calcolo dei valori hash TLSH di tutte le frasi.
- Aggiunta dell'oggetto creato al punto precedente nell'array fileDB attraverso l'utilizzo dell'operazione *push*() di javascript degli array.

Il processo appena descritto viene applicato ad ogni nome_file presente nei file json caricati al punto 1.

3. Salvataggio di un file json (trainFile.json) contenente l'array fileDB. Tale file json avrà la seguente struttura:

```
{
  data:
  [
    {
      "nomeFile": <nome_file1>,
      "sezioni": <array sezioni file 1>,
      "hashSezioni: <array digest sezioni file 1>,
      "paragrafi": <array paragrafi file 1>,
      "hashParagrafi": <array digest paragrafi file 1>,
      "frasi": <array frasi file 1>,
      "hashFrasi": <array digest frasi file 1>
     },
     ...
  ]
}
```

Listing 5.2: Struttura del json di caricamento del dataset

Grazie a tale caricamento iniziale si ottiene un unico file json contenente tutte le informazioni di cui il sistema necessita. Quindi basta utilizzare i dati salvati nel file memorizzato al punto 3 eseguendo la seguente linea di codice:

```
var fileDB=require('./train/trainFile.json').data;
```

Listing 5.3: Creazione dell'array fileDB

5.1.3 Funzionamento ricerca della similarità

Il testo inserito in input viene mandato al server attraverso tre richieste AJAX, una per ogni livello di suddivisione del testo (sezioni, paragrafi e frasi). Le richieste AJAX sono associate all'evento di click sul bottone di invio e hanno la seguente struttura:

```
    type: 'POST',
    url: '/similarity/<sezioni-paragrafi-frasi>',
    data: {
      info: <testo della textarea>,
    }
```

Listing 5.4: Richiesta AJAX per ricerca similarità

Per ogni richiesta, il server si occupa di effettuare due operazioni importanti: ristrutturazione del testo e ricerca di similarità. L'implementazione di queste operazioni rappresentano una parte cardine del progetto di tesi, esse vengono descritte in modo approfondito nella seguente sottosezione.

Ristrutturazione del testo

La ristrutturazione del testo viene fatta sul testo passato in input. Tale ristrutturazione serve per eliminare alcune caratteristiche del testo che potrebbero essere identificate come differenze tra stringhe dalla funzione di similarità. É necessario eliminare queste caratteristiche perché non sono ritenute significative per la ricerca di similarità che il sistema deve effettuare.
La azioni di ristrutturazione del testo effettuate sono le seguenti:

- testo in lowercase;
- sostituzione di newline e tabs con whitespaces;
- rimozione di spazi multipli;

- sostituzione delle abbreviazioni con la loro forma estesa.
  Le abbreviazioni considerate dal sistema sono la seguenti:
  - CEN: European Committee for Standardisation
  - EEC: European Economic Commision
  - EU: European Union
  - EC: European Commission
  - NATO: North Atlantic Treaty Organization
  - USA: United States of America
  - UK: United Kingdom
  - PGI: Principal Global Indicators
  - PDO: Protected designation of origin
  - EFSA: European Food Safety Authority

Ricerca similarità

La ricerca di similarità può essere effettuata su diversi livelli del testo: sezioni, paragrafi e frasi.
Il metodo di ricerca di similarità, a qualsiasi livello, restituisce un oggetto contenente due array:

- Array delle parti simili trovate durante la ricerca della similarità. Tale array ha la seguente struttura:
  - Chiavi array: nomi dei file della collezione di documenti per cui sono stati trovati dei match
  - Contenuto array: ogni posizione include un oggetto contenente altri tre array:
    - *partiDB*: memorizza le parti del file della collezione di documenti su cui sono state trovate similarità;
    - *partiFileNuovo*: contiene le parti del testo inserito dall'utente su cui sono state trovate similarità;

    * *livSim*: la posizione i di questo array contiene il valore di similarità calcolato da TLSH tra i digest delle stringhe: partiDB[i] e partiFileNuovo[i].

  Quindi partiDB[i] ha similarità secondo la funzione TLSH con partiFileNuovo[i].

- Array delle parti del testo.

A tutti i livelli di suddivisione del testo la ricerca della similarità funziona nel seguente modo:

1. Suddivisione del testo in base al livello richiesto utilizzando le regole illustrate precedentemente nella sezione 5.1.1. L'applicazione della regex al testo restituisce in output un array contenente le varie parti del testo ottenute.

2. Se il testo contiene almeno una parte allora per ogni parte di testo dell'array vengono effettuati i seguenti passi:

   (a) Eliminazione dei tag dell'HTML esteso.

   (b) Calcolo del digest della parte del testo considerato utilizzando la funzione *build*() dell'oggetto *new DigestHashBuilder().withHash(<hashParte>).*

   (c) Per ciascuna parte di ogni file della collezione di documenti viene effettuato il seguente processo:

      - Creazione del digest della parte della collezione di documenti utilizzando il valore di hash presente nell'array fileDB.
      - Calcolo della differenza tra il digest della parte del testo e il digest della parte della collezione di documenti che si sta considerando mediante l'utilizzo del metodo *calculateDifference()* sui due digest calcolati.
      - Se la differenza tra due digest è < 100 allora viene aggiunto un elemento all'array delle parti simili che verrà restituita. Tale inserimento avviene nel seguente modo:
        – Aggiunta nell'array partiDB della parte di testo del della collezione di documenti per cui si è trovata la similarità;

– Aggiunta nell'array partiFileNuovo della parte del testo passato come parametro per cui si è trovata la similarità.

3. Altrimenti la funzione restituisce un array contenente due array vuoti.

Nel codice ci sono 3 metodi per la ricerca della similarità, uno per ogni livello.
Le differenze tra le funzioni di ricerca di similarità dei diversi livelli si hanno nei seguenti passi di calcolo della similarità:

- Passo 1: perché vengono applicate regex diverse;
- Passo 2c: perché potrebbe essere necessario modificare il controllo dell'if al fine di migliorare i risultati della ricerca di similarità, aggiungendo ulteriori condizioni oltre al valore di differenza degli hash <100.

5.1.4 Scelte implementative test di TLSH

Il tema prevalente di tale sotto sezione riguarda le scelte implementative effettuate nell'ambiente di test di TLSH. Di seguito si riassumono tali scelte:

- Il testo passato in input viene suddiviso in sezioni, paragrafi e frasi al fine di effettuare una ricerca di similarità a diversi livelli di dettaglio. Tale suddivisione viene effettuata sfruttando la struttura dei documenti messa a disposizione dall'HTML esteso.
- Si è deciso di effettuare un caricamento iniziale del data set per evitare di calcolare più volte gli hash per tutte le parti dei file perché tale calcolo richiede molto tempo.
- Sono state attuate azioni di ristrutturazione del testo al fine di eliminare caratteristiche del testo ritenute superflue nella ricerca di similarità tra stringhe.
- Sono state implementate 3 funzioni di ricerca di similarità perché esse sono differenziate dall'inserimento di controlli specifici al livello di dettaglio (sezione, paragrafo, frase) al fine di migliorare i risultati ottenuti.

5.2 API per confronto tra funzioni LSH

In questa sezione viene illustrata la struttura delle richieste e delle risposte dell'API utilizzate per confrontare TLSH con altre implementazioni della ricerca di similarità sui documenti dell'azienda. L'API viene utilizzata dall'ambiente di confronto disponibile all'indirizzo web: http://site181928.tw.cs.unibo.it/env/.

5.2.1 Struttura della richiesta

In questa sotto sezione vengono descritte le caratteristiche che la richiesta dell'API deve soddisfare.

La richiesta per interrogare l'API correttamente deve avere la seguente struttura:

- La richiesta deve essere mandata con *method* POST;
- La richiesta deve contenere le seguenti informazioni nel campo *data*:
  - **query**: parte di testo per cui si vuole ricercare la similarità;
  - **type**: tipo o livello della ricerca che si vuole effettuare; i valori possibili sono: Section, Paragraph o Phrase;
  - **threshold**: valore minimo di similarità con la probabilità rappresentata come un numero decimale con due valori dopo la virgola;
  - **max**: numero massimo di risultati per tale richiesta.

5.2.2 Struttura della risposta

In questa sotto sezione viene descritta la struttura della risposta restituita dall'API.

La risposta restituita dall'API contiene i seguenti parametri:

- **query**: testo su cui è stata richiesta la ricerca di similarità;
- **data**: array contenente oggetti risultanti dalla ricerca di similarità. Gli oggetti di questo array hanno i seguenti attributi:

  - docname: nome del file della collezione di documenti in cui sono state trovate delle similarità;
  - text: parte del documento docname che ha similarità maggiore o uguale a threshold con il testo del campo query;
  - lev: valore della distanza di Levenshtein normalizzata calcolata tra text e query.

- **time**: tempo di ricerca dei risultati in ms;
- **max**: numero totale di risultati trovati;
- **threshold**: valore minimo che lev può assumere.

5.2.3 Gestione delle richieste

Le richieste per l'API devono essere sottoposte all'indirizzo web: http://site181901.tw.cs.unibo.it/query con method POST. Le richieste vengono successivamente intercettate e gestite dal server mediante le seguenti linee di codice:

```
app.post('/query', function(req, res) {
    var testo=digest.ristrutturazioneTesto(req.body.data)
    let ris=digest.getSim(testo, req.body.threshold,
            req.body.max,req.body.type)
    ris.query=req.body.data;
    res.setHeader('Content-Type', 'application/json');
    res.json(ris);
});
```

Listing 5.5: Gestione richieste dell'API

Il metodo *getSim()* presente nel file *gestoreDigest.js* presenta le caratteristiche sotto elencate:

- La ricerca di similarità funziona in maniera molto simile a quella descritta precedentemente nella sezione 5.1.3. L'unica differenza è rappresentata dalla struttura del risultato restituito poiché è necessario utilizzare la struttura richiesta dall'API.

- Per calcolare il tempo necessario per la ricerca dei risultati viene utilizzato il seguente metodo:
  - all'inizio della funzione viene salvato il seguente valore:

```
let start=new Date()
```

  - alla fine del calcolo della similarità viene calcolato e memorizzato nel campo *time* della risposta il tempo totale di esecuzione della ricerca con le seguenti linee di codice:

```
var end = new Date() - start
ris.time=end+"ms"
```

- Il calcolo della distanza di Levenshtein normalizzata viene effettuato dal metodo *calcolaDistanzaLevensthein()*, descritta in 2.5.

5.3 Implementazione della distanza di Levenshtein

In questa sezione viene illustrata l'implementazione del calcolo relativo alla distanza di Levenshtein descritta in 2.5. A tal proposito, il codice utilizzato è il seguente:

```
function calcolaDistanzaLevensthein(testo1, testo2){
  let n=testo1.length
  let m=testo2.length
  const d = Array(n+1).fill(null).map(()=> Array(m+1).fill(null))
  //Calcolo della prima riga e della prima colonna della matrice
  for (let i = 0; i <= n; i+=1) {
    d[i][0] = i;
  }
  for (let j = 0; j <= m; j+=1) {
    d[0][j] = j;
  }
  //Calcolo delle posizioni rimanenti della matrice
  for (let i = 1; i <= n; i += 1) {
    for (let j = 1; j <= m; j += 1) {
      const indicator = testo1[i-1] === testo2[j-1] ? 0 : 2;
      d[i][j] = Math.min(
```

```
17            d[i-1][j]+1, //eliminazione
18            d[i][j-1]+1, //inserimento
19            d[i-1][j-1]+indicator, //sostituzione
20          );
21        }
22      }
23      return Math.round((1-(d[n][m]/Math.max(n,m)))*100)/100;
24  }
```

Listing 5.6: Implementazione distanza di Levenshtein

Si noti che la funzione restituisce la distanza di Levensthein normalizzata, quindi compresa tra 0 e 1, eseguendo il seguente calcolo:

$$1 - \frac{Lev(testo1,testo2)}{Max(n,m)}$$

dove:

- Lev(testo1,testo2)=d[n][m]
- n=testo1.length
- m=testo2.length

5.4 Libreria TLSH utilizzata

L'implementazione della funzione TLSH utilizzata all'interno del progetto di tesi è disponibile nel seguente repository di github: https://github.com/idealista/tlsh-js.
Data una stringa di almeno 512 caratteri, TLSH genera un valore hash che può essere utilizzato per confronti di somiglianza. L'hash calcolato ha una lunghezza di 70 caratteri esadecimali.

5.4.1 Calcolo hash

Data una stringa, il calcolo dell'hash viene fatto attraverso il metodo *hash()* a cui deve essere passata la stringa come parametro. Il funzionamento di tale metodo richiede che la stringa abbia le seguenti caratteristiche:

- Almeno 512 caratteri;
- Una certa quantità di randomicità, per esempio nel caso in cui la stringa passata al metodo abbia il valore dei byte tutti uguali allora non verrà generato il valore hash.

Se tali requisiti sulla stringa non vengono verificati allora viene scatenata un'eccezione di tipo *InsufficientComplexityError*.

Risoluzione limite minimo caratteri

Il limite minimo di caratteri pari a 512 risulta essere molto restrittivo per gli scopi della tesi, per cercare di sottoporre alla generazione del digest anche stringhe di dimensione minore si aggiunge un padding.
Il padding utilizzato nell'implementazione dell'ambiente TLSH viene aggiunto nel seguente modo:

```
1  var stringaCasuale="puvaotihiivzyipjvitysibtnmpdrvdbuogee...";
2  function calcolaDigest(testo){
3    if(testo.length<512){
4      var len=testo.length
5      for (var z = 0; z < 512-len; z++) {
6        testo=testo+stringaCasuale[z]
7      }
8    }
9    return hash(testo)
10 }
```

Listing 5.7: Padding nella creazione del digest TLSH

Il padding viene eseguito aggiungendo una stringa fissa al testo che ha dimensione minore di 512. Se il testo venisse allungato con una stringa scelta casualmente ogni volta allora non si riuscirebbero a trovare oggetti simili nel caso di testi molto corti perché la maggior parte del digest risulterebbero diversi e quindi non si otterrebbe nessun vantaggio.

5.4.2 Calcolo differenza tra digest

Dati due valori hash $h1$ e $h2$, per calcolare la differenza tra due hash bisogna effettuare i seguenti passi:

1. Creazione di due digest, $d1$ e $d2$, usando *DigestHashBuilder* come segue:

```
var d1 = new DigestHashBuilder().withHash(h1).build();
var d2 = new DigestHashBuilder().withHash(h2).build();
```

Listing 5.8: Creazione digest TLSH di idealista

2. Ultilizzare la funzione *calculateDifference()* su d1 e d2 come segue:

```
d2.calculateDifference(d1, true);
```

Listing 5.9: Calcolo differenza tra digest TLSH di idealista

La funzione *calculateDifference()* restituisce un valore intero che ha il seguente significato:

- differenza pari a 0 indica che i due oggetti sono identici;
- differenza maggiore o uguale a 200 indica che i due oggetti sono molto diversi tra loro.

Nella parte di progetto relativa all'ambiente di test TLSH vengono considerate simili le stringhe i cui digest hanno differenza minore di 100.
Inoltre, il valore booleano passato come secondo parametro alla funzione *calculateDifference()* ha il seguente significato:

- true: include la lunghezza delle stringhe nel calcolo della differenza tra i digest;
- false: esclude la lunghezza delle stringhe nel calcolo della differenza tra i digest.

Se un input con un pattern ripetuto viene confrontato con un input con una sola istanza del pattern, la differenza risulta maggiore se viene inclusa la lunghezza utilizzando true come secondo parametro.

5.4.3 Limiti

In questa sotto sezione vengono illustrati i limiti dell'implementazione di TLSH di idealista, tali limiti comportano un abbassamento delle prestazioni del sistema e ad un alto numero di falsi negativi (vedi 6.1.1, 6.1.2).

I due principali limiti dell'implementazione di TLSH di idealista sono:

- **Ricerca lineare**: non esiste un modo per effettuare la buckettizzazione dei digest e quindi non è possibile sfruttare l'efficienza della tecnica del Locality Sensitive Hashing poiché utilizzando la funzione di confronto tra due digest (*calculateDifference()*) è possibile effettuare unicamente una ricerca lineare sul data set.
- **Limite minimo di caratteri**: il limite minimo di 512 caratteri risulta essere molto restrittivo per i nostri scopi e l'aggiunta di padding alle porzioni di testo rende più difficile la ricerca di similarità in caso di porzioni di testo molto corte.

6. Valutazione

La valutazione dell'ambiente TLSH, implementato nel progetto di tesi, avviene attraverso un'analisi quantitativa e una qualitativa. Entrambe le tipologie di test vengono illustrate nel seguente capitolo. L'analisi quantitativa ha lo scopo di esaminarne l'efficienza. Invece, la valutazione qualitativa riguarda il gradimento dell'ambiente da parte dell'utente in termini di usabilità.

6.1 Valutazione quantitativa

L'analisi quantitativa di un prodotto software ha lo scopo di testare le performance in termini di efficienza. L'efficienza di un programma può essere valutata in relazione al tempo di esecuzione, numero di risultati errati, numero di falsi positivi e falsi negativi.

In questa sezione vengono illustrati i test di efficienza dell'ambiente. Queste valutazioni sono state eseguite mediante l'utilizzo dell'ambiente di confronto, creato da Antonio Conteduca, disponibile all'indirizzo web: http://site181928.tw.cs.unibo.it/env/.
Tale ambiente utilizza l'API messa a disposizione dal sistema, descritta nella sezione 5.2.

6.1.1 Valutazione del tempo di esecuzione

La valutazione del tempo di esecuzione è stata eseguita attraverso l'invio di 20 richieste random per ogni possibile valore del campo type dell'API e in seguito sono stati analizzati i dati relativi al tempo di ricerca delle risposte ricevute dall'API.
I grafici illustrati in seguito sono stati ottenuti cliccando sul tab *Chart* dell'ambiente di confronto; essi riportano, per ogni richiesta eseguita, il tempo di esecuzione in ms della ricerca di testi simili mediante TLSH.

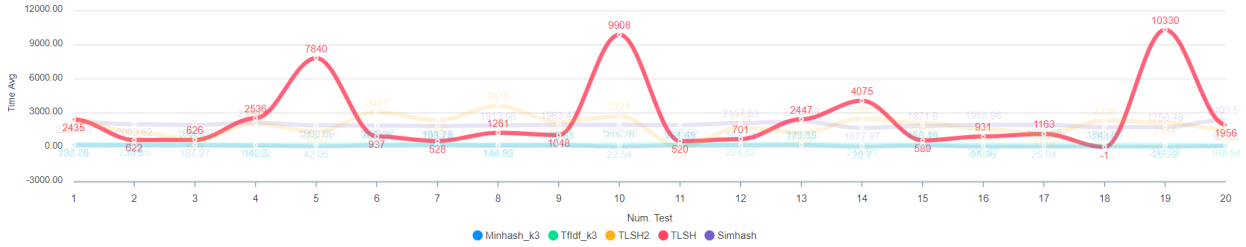

Figura 6.1: Grafico del tempo di esecuzione sulla ricerca di similarità a livello di frase.

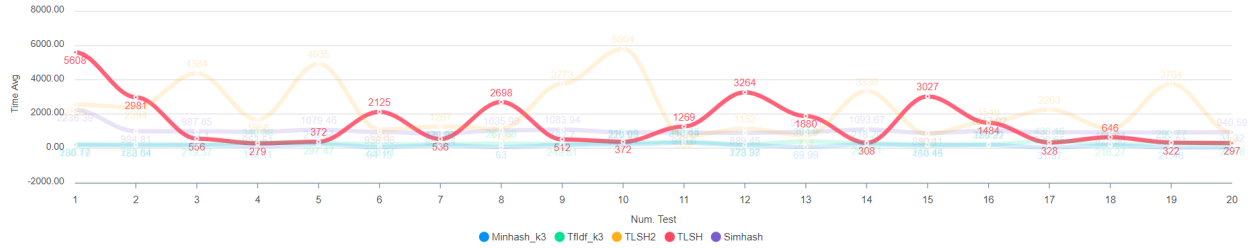

Figura 6.2: Grafico del tempo di esecuzione sulla ricerca di similarità a livello di paragrafo..

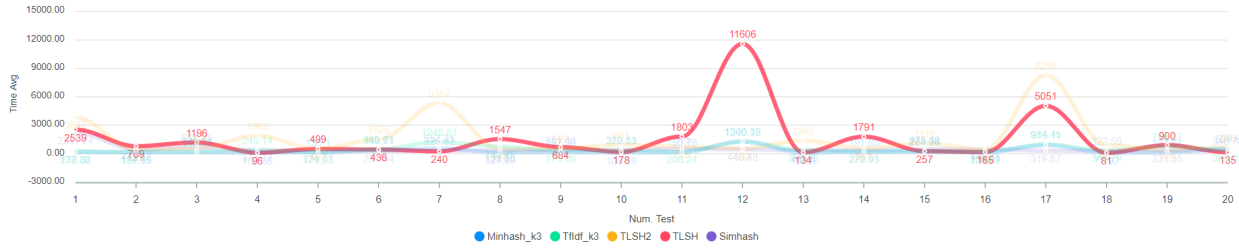

Figura 6.3: Grafico del tempo di esecuzione sulla ricerca di similarità a livello di sezione.

Dall'osservazione dei grafici sopra riportati è possibile calcolare i seguenti valori medi:

- Media di tempo per la ricerca nelle frasi: 2655 ms (escludendo il valore relativo al test 18);

- Media di tempo per la ricerca nei paragrafi: 1443 ms;
- Media di tempo per la ricerca nelle sezioni: 926 ms (escludendo il picco relativo al test 12).

Problema della ricerca lineare

Il fenomeno di abbassamento del tempo di ricerca in relazione all'aumento della dimensione della parte di testo considerata si verifica perché la ricerca di similarità viene effettuata attraverso una ricerca lineare all'interno del data set. In questo modo non vengono sfruttate le proprietà della tecnica di Locality Sensitive Hashing e si ha un costo computazionale nell'ordine di O(n).
All'interno del train data set sono presenti le seguenti quantità di dati suddivisi per sezioni, paragrafi e frasi:

Tipo	Numero
Documenti	1535
Frasi	63018
Paragrafi	30827
Sezioni	9096

Il numero di confronti che devono essere effettuati in una ricerca lineare per le frasi è molto maggiore rispetto a quello per le sezioni, di conseguenza si ha un tempo di esecuzione molto più elevato.

6.1.2 Test con distanza di Levenshtein

La distanza di Levenshtein, descritta nella sezione 2.5, viene utilizzata per avere una metrica oggettiva dei risultati e per eliminare i falsi positivi in quanto è possibile che due elementi diversi abbiano generato digest simili.

La valutazione oggettiva dell'ambiente mediate distanza di Levenshtein è stata eseguita attraverso l'invio di 20 richieste random per ogni possibile valore del campo type dell'API e in seguito sono stati analizzati i dati relativi alla media della distanza di Levenshtein delle risposte ricevute dall'API.

I grafici illustrati in seguito sono stati ottenuti cliccando sul tab *Chart* dell'ambiente di confronto; essi riportano, per ogni richiesta eseguita, il valore medio della distanza di Levenshtein dei risultati ottenuti attraverso ricerca di testi simili mediante TLSH.

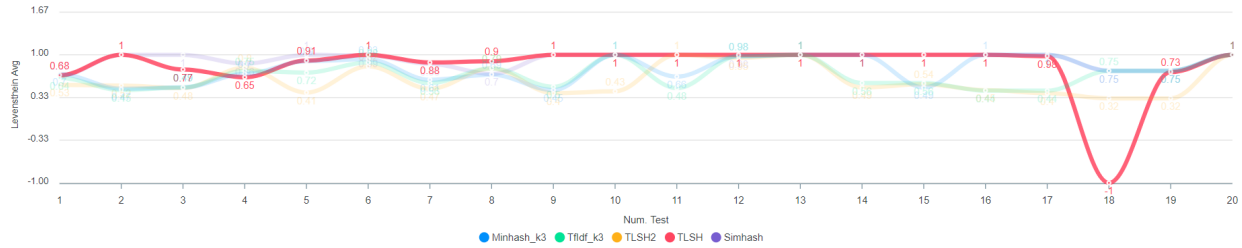

Figura 6.4: Grafico della distanza media di Levenshtein della ricerca di similarità livello di frase.

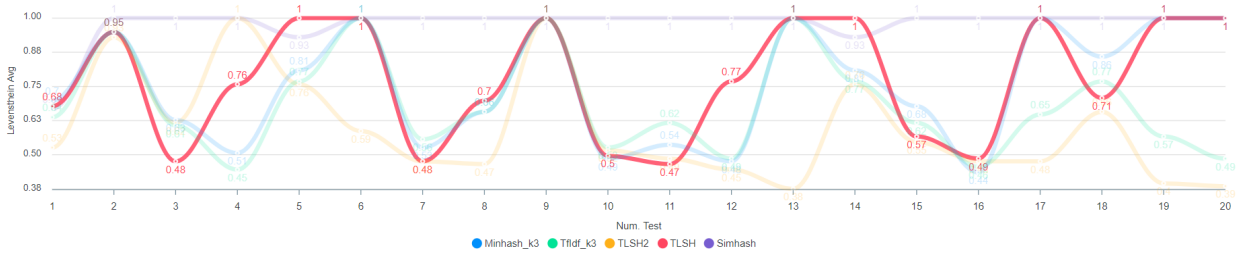

Figura 6.5: Grafico della distanza media di Levenshtein della ricerca di similarità livello di paragrafo.

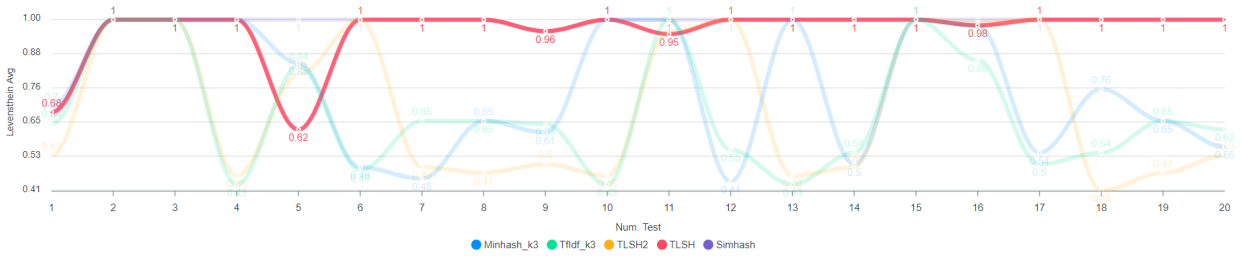

Figura 6.6: Grafico della distanza media di Levenshtein della ricerca di similarità livello di sezione.

L'algoritmo di Levenshtein calcola il numero minimo di operazioni elementari necessarie per trasformare una stringa in un'altra stringa. Di conseguenza è un calcolo della distanza tra due parti di testo molto più sensibile alle modifiche rispetto al metodo usato da TLSH. Quindi per ogni richiesta non si vuole avere delle medie delle distanze di Levenshtein pari a 1 (=100% di similarità) poiché, in tal caso, vuole dire che l'API ha restituito un solo risultato con testo equivalente a quello della query.
Dall'osservazione dei grafici sopra riportati è possibile calcolare i seguenti valori medi:

- Media della distanza di Levenshtein per la ricerca nelle frasi: 0.92 (escludendo il valore relativo al test 18);
- Media della distanza di Levenshtein per la ricerca nei paragrafi: 0.78;
- Media della distanza di Levenshtein per la ricerca nelle sezioni: 0.96.

Si noti che nei casi delle frasi e delle sezioni la media della distanza di Levenshtein è molto vicina a 1. I valori più alti dei grafici 6.4 e 6.6 si sono verificati nelle seguenti situazioni: dimensione del testo della query molto minore di 512 caratteri, dimensione del testo della query molto maggiore di 512 caratteri. La prima situazione potrebbe essere causata dal padding usato in modo di oltrepassare il limite minimo di caratteri imposto dall'implementazione di TLSH di idealista, come descritto nella sezione 5.4. Diversamente, la seconda situazione potrebbe essere causata dalla scelta del threshold 100 per TLSH nell'implementazione della ricerca di similarità (vedi sezione 5.1.3); in alcuni casi tale soglia risulta troppo restrittiva.

6.2 Valutazione qualitativa

L'analisi qualitativa di un prodotto software ha come obiettivo quello di valutare l'efficacia del prodotto. L'efficacia di un prodotto software può essere misurata in base alla facilità di utilizzo e alla capacità di rendere l'utente in grado di usare tutte le funzionalità del prodotto nel minore tempo; tale proprietà è nota come *Learnability*. Inoltre l'efficacia di un programma valuta la soddisfazione dell'utente durante l'utilizzo del prodotto.

6.2.1 Questionario SUS (System Usability Scale)

Il System Usability Scale [LS09] [SUS] fu ideato da John Brooke nel 1986 come test generico e veloce per valutare la soddisfazione dell'utente. Il SUS è un metodo definito da un protocollo fisso e da un criterio standard di valutazione. Tale procedimento di valutazione è indipendente dalla tecnologia, infatti è stato utilizzato come test in vari ambiti, tra i quali: hardware, applicazioni mobili, siti web e telefoni cellulari.

Utilizzo del questionario SUS

Il questionario SUS è composto da 10 affermazioni, che si alternano tra affermazioni positive e negative (dispari=positive e pari=negative):

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

Le affermazioni 4, 8 e 10 riguardano la proprietà di learnability del sistema, mentre le affermazioni 1, 2, 3, 5, 6, 7, 8 e 9 si riferiscono alla proprietà di usabilità.

L'utente, a cui viene richiesta la compilazione del questionario, deve rispondere ad

ogni affermazione con valori compresi tra 1 e 5, dove 1 equivale a forte disaccordo con l'affermazione e 5 corrisponde ad una forte accordo con l'affermazione.

Calcolo e interpretazione dei risultati di SUS

Il calcolo dei risultati SUS avviene attraverso il seguente processo:

1. Per le affermazioni con numero dispari si sottrae 1 dalla risposta dell'utente (<risposta utente> -1).
2. Per le affermazioni con numero pari si sottrae la risposta dell'utente da 5 (5$-$ <risposta utente>).
3. Si sommano i valori di ogni utente e si moltiplica per 2.5. In questo modo si converte l'intervallo di valori possibili da 0 a 100 invece che da 0 a 40.
4. Si calcola la media dei valori ottenuti per ogni utente, questo valore è chiamato punteggio SUS. Un punteggio SUS maggiore o uguale a 68 rappresenta una buona usabilità del sistema.

6.2.2 Test usabilità con SUS

Dall'applicazione delle basi teoriche del System Usability Scale, descritte nella sottosezione 6.2.1, si riportano i dati raccolti relativi alla sottomissione di tale questionario ad un numero di utenti pari a 5 e se ne calcolano i risultati. Infine, si riporta il punteggio SUS ottenuto.

Dati raccolti

Utente	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**
Utente1	5	1	5	1	4	1	4	1	4	1
Utente2	3	1	4	3	4	1	4	1	3	1
Utente3	5	1	4	1	4	1	5	2	5	1
Utente4	5	1	5	2	4	1	5	2	4	1
Utente5	5	1	5	3	4	1	3	2	5	1

Tabella 6.1: Tabella risposte questionario SUS

Calcolo risultati del test

L'applicazione del processo di calcolo dei risultati del test SUS, descritto nella sezione 6.2.1, ci permette di ricavare i seguenti valori:

Utente	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Somma*2.5**
Utente1	4	4	4	4	3	4	3	4	3	4	37*2.5=92.5
Utente2	2	4	3	2	3	4	3	4	2	4	31*2.5=77.5
Utente3	4	4	3	4	3	4	4	3	4	4	37*2.5=92.5
Utente4	4	4	4	3	3	4	4	3	3	4	36*2.5=90
Utente5	4	4	4	2	3	4	2	3	4	4	34*2.5=85

Tabella 6.2: Tabella risultati questionario SUS

Si ottiene un punteggio SUS pari a 87.5.

7. Conclusioni e sviluppi futuri

Questo studio si poneva come obiettivo lo sviluppo di un sistema di ricerca, raggruppamento e classificazione di somiglianza basato sulla funzione TLSH che fosse efficiente e preciso al fine di far fronte a problemi legati alla ricerca di testi simili all'interno di una collezione di documenti aziendali di grandi dimensioni. Dall'osservazione dei risultati ottenuti è possibile affermare che l'esito del progetto è risultato soddisfacente e ha portato a considerazioni utili nel campo della ricerca dei Nearest Neighbors.

L'analisi quantitativa riguardante l'efficienza computazionale ha prodotto esiti modesti nel caso di ricerca di similarità a livello di frase. Tali esiti sono causati dalla ricerca lineare utilizzata nello studio della similarità.
Il miglioramento degli esiti relativi all'efficienza computazionale riguardano l'ideazione un metodo di buckettizzazione dei digest all'interno tabella hash al fine di sfruttare le proprietà delle tecniche di Locality Sensitive Hashing.
L'analisi quantitativa inerente alla precisione della ricerca di similarità a livello di frase e di sezione ha generato esiti insoddisfacenti. Quest'ultimi, nel caso del livello di frase, sono dovuti al padding utilizzato per superare il limite minimo di caratteri imposto dall'implementazione di TLSH adoperata nel progetto. Il perfezionamento dell'ambiente, volto ad evitare tali falsi negativi, è ottenibile unicamente mediante l'utilizzo di un'altra implementazione della funzione TLSH.
Gli esiti discreti ottenuti nel caso del livello di sezione sono causati dalla soglia, determinata sperimentalmente, pari a 100 impiegata nella ricerca di oggetti simili con TLSH. Essa risulta essere troppo restrittiva quando si considerano sezioni di dimensione molto grande.
Tuttavia, l'analisi quantitativa dell'efficienza dell'ambiente negli altri casi ha prodotto esiti eccellenti.
Un miglioramento nella precisione dei risultati si potrebbe ottenere aggiungendo ulteriori

controlli in fase di ristrutturazione del testo (vedi 5.1.3) al fine di eliminare altre parti del testo che non sono ritenute significative per la ricerca di similarità.
L'analisi qualitativa, finalizzata alla valutazione della soddisfazione dell'utente in termini di usabilità, ha prodotto un punteggio SUS pari a 87.5 molto più alto del valore medio, identificato nel valore 68.

In conclusione, TLSH si è dimostrata una buona funzione di ricerca degli oggetti simili e dunque con applicabilità notevole al problema dei Nearest Neighbors.
Sviluppi futuri riguardanti questa funzione potrebbero concernere l'ideazione di un metodo di buckettizzazione dei digest e l'eliminazione del limite minimo di caratteri richiesti per la creazione del digest.

8. Bibliografia

8.1 Articoli

[Bay+09] Ulrich Bayer et al. ≪Scalable, Behavior-Based Malware Clustering≫. In: *CiteSeerX* (2009). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.148.7690&rep=rep1&type=pdf.

[Bre+14] Frank Breitinger et al. ≪ Approximate Matching: Definition and Terminology ≫. In: *NIST* (2014). DOI: 10.6028/NIST.SP.800-168.

[CM15] Wei Cen e Kehua Miao. ≪An Improved Algorithm for Locality-Sensitive hashing≫. In: *IEEE* (2015). DOI: 10.1109/ICCSE.2015.7250218.

[De +04] S. De Capitani di Vimercati et al. ≪An Open Digest-based Technique for Spam Detection≫. In: *CiteSeerX* (2004). URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.61.6185.

[HBB16] Vikram S. Harichandran, Frank Breitinger e Ibrahim Baggili. ≪Bytewise Approximate Matching: The Good, The Bad, and The Unknown≫. In: *The Association of Digital Forensics, Security and Law (ADFSL)* 11 (2016). DOI: 10.15394/jdfsl.2016.1379.

[Hea03] Marti Hearst. ≪What Is Text Mining? ≫ In: (2003). URL: https://www.jaist.ac.jp/~bao/MOT-Ishikawa/FurtherReadingNo1.pdf.

[Kor06] Jesse Kornblum. ≪Identifying almost identical files using context triggered piecewise hashing≫. In: *ACM* 3 (2006), pp. 91–97. DOI: 10.1016/j.diin.2006.06.015.

[LS09] James R. Lewis e Jeff Sauro. ≪The Factor Structure of the System Usability Scale≫. In: *ResearchGate* (2009). DOI: 10.1007/978-3-642-02806-9_12.

[MH17] Vitor Hugo Galhardo Moia e Marco Aurélio Amaral Henriques. ≪Similarity Digest Search: A Survey and Comparative Analysis of Strategies to Perform Known File Filtering Using Approximate Matching≫. In: *Hindawi* (2017). DOI: 10.1155/2017/1306802.

[OCC13] Jonathan Oliver, Chun Cheng e Yanggui Chen. ≪TLSH - A Locality Sensitive Hash≫. In: *IEEE* (2013). DOI: 10.1109/CTC.2013.9.

[OFC14] Jonathan Oliver, Scott Forman e Chun Cheng. ≪Using Randomization to Attack Similarity Digests≫. In: *Springer* 490 (2014). DOI: 10.1007/978-3-662-45670-5_19.

[Pea90] Peter K. Pearson. ≪Fast hashing of variable-length text strings≫. In: *ACM* 33 (1990), pp. 677–680. DOI: 10.1145/78973.78978.

[Pry06] Alexey Pryakhin. ≪Similarity Search and Data Mining Techniques for Advanced Database Systems≫. In: (2006). URL: https://edoc.ub.uni-muenchen.de/6375/1/Pryakhin_Alexey.pdf.

[Ras+09] Venkat Rasagna et al. ≪Robust Recognition of Documents by Fusing Results of Word Clusters≫. In: *IEE* (2009). DOI: 10.1109/ICDAR.2009.135.

[Rou10] Vassil Roussev. ≪Data Fingerprinting with Similarity Digests≫. In: *Springer* 337 (2010). URL: https://link.springer.com/content/pdf/10.1007/978-3-642-15506-2_15.pdf.

[Rou11] Vassil Roussev. ≪An evaluation of forensic similarity hashes≫. In: *ScienceDirect* (2011). DOI: 10.1016/j.diin.2011.05.005.

[SC08] Malcom Slaney e Michael Casey. ≪Locality-Sensitive Hashing for Finding Nearest Neighbors≫. In: *IEEE* 25 (2008), pp. 128–131. DOI: 10.1109/MSP.2007.914237.

8.2 Libri

[BM14] Alan A. Bertossi e Alberto Montresor. *Algoritmi e strutture di dati*. 3ª ed. CittàStudi, 2014.

[JM18] Daniel Jurafsky e James H. Martin. *Speech and Language Processing*. 3ª ed. 2018.

[LRU14] Jure Leskovec, Anand Rajaraman e Jeffrey D. Ullman. *Mining of Massive Datasets*. 3ª ed. Oxford University, 2014.

[Rez13] Alessandro Rezzani. *Big Data, Architettura, tecnologie e metodi per l'utilizzo di grandi basi di dati*. Apogeo Editore, 2013.

8.3 Siti Web consultati

[balsamiq] *Software Balsamiq*. URL: `https://balsamiq.com/`.

[BioNLP] *Biomedical Text Mining (BioNLP) su Wikipedia*. URL: `https://en.wikipedia.org/wiki/Biomedical_text_mining`.

[BioNLP2] *Linking genes to literature: text mining, information extraction, and retrieval applications for biology*. URL: `https://genomebiology.biomedcentral.com/articles/10.1186/gb-2008-9-s2-s8`.

[ciscoblog] *Malware Validation Techniques su Cisco Blogs*. URL: `https://blogs.cisco.com/security/malware_validation_techniques`.

[ssdeep] *Implementazione ssdeep*. URL: `https://ssdeep-project.github.io/ssdeep/`.

[SUS] *Measuring Usability with the System Usability Scale*. URL: `https://measuringu.com/sus/`.

[TextMining] *Text mining su Wikipedia*. URL: `https://en.wikipedia.org/wiki/Text_mining`.

[TLSH] *Implementazione TLSH su GitHub*. URL: `https://github.com/idealista/tlsh`.

[trendmicroblog] *Smart Whitelisting Using Locality Sensitive Hashing su TrendMicro Blog*. URL: https://blog.trendmicro.com/trendlabs-security-intelligence/smart-whitelisting-using-locality-sensitive-hashing/.

[wikiNN] *Nearest neighbor search su Wikipedia*. URL: https://en.wikipedia.org/wiki/Nearest_neighbor_search.