Alma Mater Studiorum · Università di Bologna

Scuola di Scienze
Corso di Laurea Magistrale in Informatica

L'uso di tecniche di similarità nell'editing di documenti fortemente strutturati

Relatore:
Chiar.mo Prof. Fabio Vitali

Presentata da:
Antonio Conteduca

Anno Accademico 2018-19
Sessione II

"La pura ricerca è quando faccio quello che non so fare."

Werner von Braun

Contenuti

Capitolo 1

Introduzione

Ogni giorno il numero di documenti prodotti è in continua crescita e con ciò aumenta sempre più la necessità di trovare tecniche e metodologie che siano in grado di recuperare gli elementi più rilevanti sulla base di una qualche nozione di similarità. In virtù di questa necessità, la corrente dissertazione ha l'obiettivo di verificare la tesi secondo cui *l'utilizzo di metodologie basate sul calcolo della similarità può migliorare l'editing di documenti strutturati attraverso attività di ricerca, recupero e confronto che forniscono supporto testuale a vari livelli di dettaglio.* Immaginiamo di avere una collezione di documenti e di volerne scrivere un nuovo che si armonizzi con il contenuto e la struttura dei documenti già esistenti. Immaginiamo poi di avere un sistema che, osservando il nuovo documento durante la fase di editing, sia in grado di fornire suggerimenti testuali che siano simili al testo del nuovo documento e che derivino da una precedente analisi sugli elementi della collezione. Spetta successivamente all'utente decidere se tali suggerimenti possano essere utili o meno. Il corrente progetto di tesi è quindi orientato alla creazione del suddetto sistema il cui scopo è quello di fornire all'utente una serie di suggerimenti testuali (come frasi, paragrafi e sezioni) che possano facilitare la fase di editing conformando il nuovo documento a quelle linee guida presenti nella data collezione di documenti strutturati.

In alcuni ambiti è necessario ottenere una certa forma di similarità e omogeneità tra testi e perciò progettare e implementare un sistema così fatto può migliorare e velocizzare la fase di scrittura di nuovi documenti.

Ricercare documenti simili assume molta importanza soprattutto nell'ambito legislativo, ove la similarità permette di semplificare le attività relative la creazione di un nuovo documento. Il continuo confronto con documenti legali già esistenti è un compito che garantisce ai nuovi documenti più omogeneità e integrità oltre a renderli più facili da leggere e applicare. Il collegamento di documenti correlati garantisce, quindi, che situazioni simili siano trattate e descritte allo stesso modo.

Il concetto di similarità è abbastanza noto in letteratura. La similarità tra parole, frasi, paragrafi e documenti, anche conosciuta come *text similarity*, è una componente importante in molti campi come, ad esempio, l'*information retrieval*, clustering di documenti, disambiguazione del significato delle parole, machine translation e text summarization. La text similarity ha molte applicazioni importanti come, ad esempio, nei motori di ricerca che necessitano di rilevare i risultati da proporre all'utente in base ai bisogni di quest'ultimo oppure nei sistemi di domande e risposte come *Quora*[1] o *Stack Overflow*[2] in modo da evitare la duplicazione di domande già poste in precedenza.

Molti sono gli studi su tecniche che affrontano il problema della text similarity. Ognuna di queste tecniche condivide l'idea di dover trasformare i documenti in *vettori di features* e confrontare questi attraverso delle misure che definiscono la similarità tra le relative rappresentazioni. La generazione di una qualche forma di rappresentazione è una delle componenti essenziali per permettere la costruzione di un sistema efficiente in termini di tempi di risposta e precisione dei risultati.

La situazione su cui ci concentreremo nel seguente lavoro di tesi prevede principalmente tre fasi:

1. l'utente del sistema redige un nuovo documento con l'obiettivo di renderlo conforme agli elementi della collezione;

2. il sistema "osserva" il contenuto del nuovo documento durante l'intera fase di editing con l'obiettivo di ricercare frammenti di testo simili al

[1] *Quora* è una community online in cui poter ottenere risposte o dibattiti su qualunque tema, https://it.quora.com/

[2] *Stack Overflow* è un community online di sviluppatori software, https://stackoverflow.com/

testo inserito per poi suggerirli all'utente;

3. l'utente, durante la fase di editing, può decidere se utilizzare il suggerimento che meglio si abbina al contenuto del nuovo documento, apportando, eventualmente, modifiche al suggerimento stesso, oppure, può semplicemente ignorare gli "aiuti" forniti dal sistema.

Come vedremo più in avanti, la soddisfazione dell'utente è una componente fondamentale per comprendere al meglio l'efficienza del sistema. Le logiche che determinano se un suggerimento è o non è rilevante dipendono dalla tipologia di text similarity considerata che può lessicale o semantica [GF13]. L'obiettivo del progetto di tesi è ricercare frammenti di testo appartenenti alla collezione di documenti iniziale in modo da aiutare, velocizzare e facilitare l'utente durante la fase di scrittura di un nuovo documento attraverso un'analisi orientata alla similarità lessicale.

Per tale motivo, si ha la necessità di sviluppare delle tecniche che ci permettano di ricercare, in tempi ragionevoli, i frammenti di testo più rilevanti da suggerire all'utente sulla base del testo fornito durante la fase di editing. Una possibile soluzione potrebbe essere quella di applicare un confronto uno a uno tra il testo in input e i documenti della collezione. Scelta una metrica di similarità sulla base della logica adottata, si procede andando a considerare solo quegli elementi che soddisfano una certa soglia di similarità. Quest'ultimo sarà l'argomento principale del **capitolo 2** in cui verranno discusse alcune metriche di similarità che consentono di calcolare un valore numerico che identifica il grado di somiglianza tra due testi sia in ambito lessicale che semantico.

Con la semplice nozione di similarità, però, siamo in grado solamente di applicare una ricerca lineare, non accettabile con una collezione composta da migliaia di documenti. Potremmo pensare di abbassare il tempo di computazione impiegato per calcolare il valore di similarità ma rimarrebbe l'inefficienza derivante dalla ricerca brute force.
Durante il corso del **capitolo 3** verranno presentate varie metodologie che permettono di rappresentare in uno spazio vettoriale un testo così da permettere l'applicazione di operazioni algoritmiche. Tale processo è chiamato *vectorization* e consente di ridurre la dimensionalità degli elementi e il tempo

di calcolo necessario per il confronto tra questi. La proprietà principale che tale rappresentazione deve possedere è quella di preservare la similarità: se due documenti sono simili in base a un valore di similarità ci aspetteremmo che, anche dopo il processo di vectorization, tale valore rimanga più o meno invariato.

Il problema di ricerca di frammenti simili può essere formalizzato attraverso il problema dei *Nearest Neighbors*[3]. Concettualmente tale problema è facilmente risolvibile effettuando una ricerca lineare ma, come già detto, una soluzione di questo genere non è scalabile con un ingente numero di documenti. Questo sarà l'argomento principale del **capitolo 4** in cui verrà descritta una soluzione efficiente ottenibile attraverso un rilassamento del problema NN, chiamato *Approximate Nearest Neighbor*, permettendo di ricavare l'insieme dei documenti più simili (dopo una fase di vectorization) rispetto ad un determinato punto, *query point*, e di una metrica di similarità definita inizialmente.

Una delle tecniche più note per la risoluzione dell'Approximate Nearest Neighbor è la *Locality Sensitive Hashing (LSH)*. LSH è una famiglia di funzioni che permette di raggruppare oggetti simili partendo da una collezione di elementi e dalla definizione di una metrica di similarità. L'intuizione dietro LSH, il cui utilizzo verrà discusso nel dettaglio nel corso di questo elaborato, è quella di utilizzare una serie di funzioni hash in modo da massimizzare la probabilità per documenti simili di essere mappati nello stesso gruppo, chiamato *bucket*. Ad ogni documento della collezione verranno applicate delle operazioni di normalizzazione il cui scopo è quello di preprocessare il documento e successivamente creare delle *signatures* tramite l'algoritmo *Minhash* (un'istanza di LSH), che mantenga quella proprietà, già detta prima, di salvaguardia della similarità.
Ogni variante di LSH deve poter stimare il valore di una determinata metrica di similarità. Vedremo come le signatures prodotte da Minhash siano una stima di una nota metrica di similarità, chiamata *indice di Jaccard*, che consente di quantificare il grado di somiglianza tra due insiemi. Oltre questo approccio, verrà descritta un'ottimizzazione della tecnica TF-IDF che per-

[3] *Nearest neighbor search*, https://en.wikipedia.org/wiki/Nearest_neighbor_search

mette di ridurre la dimensione del vettore generato evitando una dimensione proporzionale alla cardinalità dell'insieme di token.

Nel **capitolo 5** verrà fornita una descrizione dettagliata riguardante il contributo offerto in questo lavoro di tesi che include: l'implementazione di due tecniche quali Minhash e MRCTA; la progettazione dell'ambiente SHE (Similarity Hashing Environment) che offre le funzionalità necessarie per consentire le fasi di sviluppo e testing di una qualsiasi altra tecnica; la definizione di un modello comune per permettere un confronto all'interno di SHE tra le varie tecniche attualmente disponibili e dare la possibilità a quelle future di poter integrarsi in maniera rapida e veloce.

Si discuteranno poi i test e le valutazioni delle tecniche sulla base di un'analisi che mira ad identificare il grado di soddisfazione dell'utente. Nell'analisi soggettiva, che verrà descritta nel **capitolo 6**, l'attenzione verrà spostata sull'esperienza dell'utente. L'esperienza utente è una valutazione soggettiva in quanto ognuno di questi interagisce e sperimenta in base al proprio punto di vista e sulla base di specifici obiettivi.

La dissertazione si concluderà con la presentazione di un riepilogo del lavoro svolto e degli argomenti trattati nel corso dei vari capitoli, seguito da potenziali sviluppi futuri rivolti a chi auspicabilmente continuerà ed espanderà questo progetto di tesi.

Capitolo 2

Text Similarity

Il concetto di similarità tra testi copre un ruolo fondamentale nella dimostrazione della corrente dissertazione quindi, prima di addentrarci sul come sia possibile raggiungere l'obiettivo prefissato, è giusto fornirne una chiara ed esaustiva definizione. Nel seguente capitolo andremo a vedere due diversi tipi di similarità testuale, lessicale e semantica, ponendo l'attenzione principalmente sulla similarità lessicale e guardando anche alcuni aspetti che introducono il calcolo della similarità semantica. Si analizzeranno, successivamente, alcune metodologie che ci permettono di quantificare i due tipi di similarità.

2.1 Cos'è la text similarity?

Avere dei meccanismi per misurare la *text similarity* è un buon punto di partenza per la ricerca di similarità in collezioni di documenti di considerevoli dimensioni. Trovare la somiglianza tra parole è un passo primario per determinare la similarità tra frasi, paragrafi e documenti. Un approccio di questo tipo può facilitare la ricerca di informazioni più pertinenti. Questo è un fondamento importante nell'ambito del data mining così come nell'*Information Retrieval*, classificazione del testo, estrazione di informazioni, clustering di documenti, sentiment analysis, machine translation, text summarization e *Natural Language Processing (NLP)*. Possiamo definire la similarità come una funzione *SIM(x,y)* che associa alla coppia in input un

valore compreso tra 0 e 1. Un valore pari a 0 indica una totale assenza di similarità tra i due oggetti in input mentre un valore pari a 1 indica una perfetta similarità. Alcune delle tecniche per la misurazione della similarità che vedremo nel capitolo corrente, prevedono una trasformazione del testo che consiste nel rappresentare il documento in un *vettore di features*. In tal modo, avendo un rappresentazione matematica del testo saremo in grado di calcolare la distanza o similarità dei documenti attraverso l'utilizzo della loro rappresentazione.

$$SIM : (x, y) \rightarrow [0, 1] \quad con\ x, y \in \mathbb{R}^d$$

Due documenti, o in generale testi, possono essere simili in due modi: lessicalmente e semanticamente [GF13]. I testi sono lessicalmente simili se hanno una sequenza di caratteri in comune. Se invece hanno lo stesso tema, allora diremo che sono semanticamente simili.Per esempio, la coppia di parole ("book","cook") avrà un alto grado di similarità lessicale, ma semanticamente non indicano lo stesso concetto. La coppia ("car", "wheel") non ha alcuna similarità lessicale ma semanticamente sono molto correlate in quanto entrambe le parole indicano termini dell'ambito automobilistico. Sebbene i metodi per la similarità lessicale siano spesso usati per ottenere quella semantica, in un certo senso, ottenere una perfetta comprensione semantica del testo è molto complicato.

La selezione di documenti rilevanti attraverso una qualche metrica di similarità è fondamentale in applicazioni che lavorano con grosse quantità di dati. Sono presenti varie metriche di similarità in letteratura ma non esiste oggettivamente una migliore di un'altra [VK16]. Mentre alcune misure sono state studiate e valutate singolarmente, esse non sono adeguate per risolvere problemi del mondo reale. Un crescente numero di problemi relativi specialmente alle tecnologie di ricerca nel web dipendono fortemente da un accurato calcolo della similarità. Trovare la più adatta è spesso un compito molto cruciale.

Alcuni linguaggi di programmazione dispongono di funzioni *built-in* che lavorano in maniera efficiente nel calcolare l'esatta somiglianza tra due stringhe ma nel caso in cui ci sia bisogno di conoscere una qualche relazione di somiglianza quantificabile, tali funzioni non possono essere più utili al nostro

scopo. Non esiste, quindi, un modo nei linguaggi di programmazione che ci permetta di dire se un documento è simile al 60 percento con un altro documento.
Un altro problema si presenta quando l'utente ha bisogno di trovare delle informazioni che soddisfino un certo predicato di similarità in quanto ottenere il risultato esatto non è sufficiente. Questo tipo di richieste sono molto importanti nell'ambito delle applicazione web dove gli errori dovuti alle abbreviazioni sono davvero comuni.
Molte sono le metriche esistenti ma l'utilizzo di una rispetto ad un'altra può dipendere spesso dal particolare dominio di applicazione.

Prima di iniziare con le differenze tra i due vari tipi di similarità e le varie tecniche per quantificarla è doveroso chiarire il significato di due parole chiave: *similarità* e *distanza*. Diremo che:

- due documenti hanno *distanza* 1 quando sono differenti e hanno distanza 0 quando sono perfettamente simili;
- due documenti hanno *similarità* pari a 1 quando sono perfettamente simili e hanno similarità pari a 0 quando sono differenti;

È sempre possibile ottenere l'una dall'altra. Siano D_1 e D_2 due vettori rappresentanti due documenti e SIM la funzione che calcola la similarità e $Dist$ la funzione per la distanza. Quindi:

- $SIM(D_1, D_2) = 1.0 - Dist(D_1, D_2)$
- $Dist(D_1, D_2) = 1.0 - SIM(D_1, D_2)$

2.2 Similarità lessicale

Il concetto di text similarity può essere visto in base a due livelli di comprensione: lessicale e semantica. In questa sezione, e nel corso della seguente dissertazione, ci focalizzeremo maggiormente sugli aspetti relativi a quella lessicale. Per similarità lessicale si intende il grado di somiglianza che intercorre (word by word) tra due testi. Per esempio, cosa possiamo dire delle similarità lessicale di "*il gatto ha mangiato il topo*" con "*il topo ha mangiato il cibo del gatto"* guardando semplicemente le parole? Se ci concentriamo solo a questo livello, le frasi appaiono molto simili, infatti 3 parole su 4 sono condivise (ignorando gli articoli e le preposizioni).

$$\textit{"gatto ha mangiato topo"} \cup \textit{"topo ha mangiato cibo gatto"} = 3$$

Questa nozione è spesso riferita alla similarità lessicale. Sebbene il significato reale delle frasi sia ignorato, ciò non toglie che tale nozione di similarità sia inefficace. Si possono trovare modi efficienti per espandere la portata di ogni parola o un insieme di parole per migliorare la somiglianza tra due frammenti di testo che si vuole confrontare.

Vediamo ora alcune delle metriche che ci permettono di determinare la similarità lessicale tra due testi. In tali metriche viene indicato con 1 la similarità perfetta e con 0 la totale diversità.

Longest common subsequence (LCS) LCS[1] è una tecnica comunemente usata per misurare la similarità tra due stringhe *(X, Y)* ed è alla base del programma *diff* (un software di comparazione di file). LCS misura la più lunga sotto sequenza comune alle due stringhe. Si noti che una sotto sequenza non è necessariamente una sotto stringa. Ad esempio:

$$\text{LCS}(\text{“}\textit{Facebook}\text{”},\text{“}\textit{Google}\text{”}) = 2$$
in quanto *"oo"* è la sotto sequenza comune.

Siano $X = (x_1, x_2, ..., x_N)$ e $Y = (y_1, y_2, ..., y_N)$ due stringhe. Sia $LCS(X, Y)$ il valore della sotto sequenza comune calcolata nel seguente modo:

[1] *Longest common subsequence problem*,
https://en.wikipedia.org/wiki/Longest_common_subsequence_problem

$$LCS(X,Y) = \begin{cases} 0 & se\ i = 0\ oppure\ j = 0 \\ 1 + LCS(X_{i-1}, Y_{j-1}) & se\ X_i = Y_j \\ max(LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)) & se\ X_i \neq Y_j \end{cases}$$

Tale algoritmo ha complessità pari a $O(n)$ ed è basato sulla tecnica della programmazione dinamica. Se vogliamo ottenere un valore compreso tra 0 e 1 (dove 1 indica la similarità perfetta) dobbiamo dividere il valore ottenuto da LCS con la lunghezza massima tra le due stringhe:

- $LCS_{norm}(\text{“}Facebook\text{”}, \text{“}Google\text{”}) = 2/8 = 0.25$
- $LCS_{norm}(\text{“}Facebook\text{”}, \text{“}Facebook\text{”}) = 8/8 = 1.00$

Levenshtein distance La distanza di Levenshtein [2] è una tecnica utilizzata per definire la distanza tra due stringhe contando il numero minimo di operazioni necessarie per trasformare una stringa nell'altra. Le operazioni definite comprendono gli inserimenti, le cancellazioni e le sostituzioni di un singolo carattere. Ad esempio, per trasformare la parola “*Google*” in “*Facebook*” occorre:

1. sostituire la G con la F in posizione 1 → *Foogle*
2. inserire la a in posizione 1 → *Faoogle*
3. inserire la c in posizione 1 → *Facoogle*
4. inserire la e in posizione 1 → *Faceoogle*
5. inserire la b in posizione 1 → *Faceboogle*
6. sostituire la G con la k in posizione 4 → *Facebookle*
7. cancellare la l in posizione 5 → *Facebooke*

[2] *Levenshtein Distance Algorithm*, https://dzone.com/articles/the-levenshtein-algorithm-1

8. cancellare la e in posizione 6 → *Facebook*

Complessivamente, sono necessarie 8 operazioni. Pertanto, in questo esempio la distanza di Levenshtein è uguale 8. La distanza di Levenshtein tra due stringhe $X = (x_1, x_2, ..., x_N)$ e $Y = (y_1, y_2, ..., y_N)$ è data da:

$$LEV(X_i, Y_j) = \begin{cases} max(i,j) & se\ min(i,j) = 0 \\ min \begin{cases} LEV(X_{i-1}, Y_j) + 1 \\ LEV(X_i, Y_{j-1}) + 1 \\ LEV(X_{i-1}, Y_{j-1}) + 1 \end{cases} & altrimenti \end{cases}$$

Tale tecnica ha complessità media pari a $O(n+d^2)$, dove n è la lunghezza della stringa più lunga e d è il numero di operazioni necessarie. In tal modo si ottiene una distanza ossia un numero intero che indica le operazioni necessarie affinché le due stringhe in input diventino una stessa stringa. Come nel caso della metrica precedente, se vogliamo ottenere un valore compreso tra 0 e 1 (dove 1 indica la similarità perfetta) dobbiamo applicare la seguente formula:

$$LEV_{norm}(X_i, X_j) = 1 - \frac{LEV(X_i, Y_j)}{max(|X|, |Y|)}$$

Quindi:

- $LEV_{norm}(\text{“}Facebook\text{”}, \text{“}Google\text{”}) = 1 - 8/8 = 0.0$
- $LEV_{norm}(\text{“}Facebook\text{”}, \text{“}Facebook\text{”}) = 1 - 0/8 = 1.0$

Euclidean distance Per ottenere tale similarità ogni documento necessita di essere trasformato in un vettore (vedremo varie tecniche di come applicare tale dimostrazione nel capitolo 3). Ogni parola del testo definisce una dimensione del vettore nello spazio Euclideo e la frequenza di ogni parola corrisponde al valore della dimensione. Euclidean distance, o *L2 distance*, è calcolata applicando la seguente formula:

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2} = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$$

dove q e p sono due documenti rappresentati sotto forma di vettore.

	Dhoni	*Cricket*	*Sachin*
Doc A	10	50	200
Doc B	400	100	20
Doc C	10	5	1

Tabella 2.1: Numero di occorrenze delle tre parole considerate

Cosine similarity Un comune approccio usato per confrontare testi è basato sulla conta del numero massimo di parole comuni tra i due testi ma tale approccio presenta un grande svantaggio: con l'aumentare della dimensione del testo, il numero di parole comuni tende ad aumentare anche se i testi non sono correlati. La *cosine similarity*[3] aiuta a risolvere questo svantaggio determinando quanto i testi siano simili indipendentemente dalla loro dimensione. Anche in questo caso si necessita di una trasformazione vettoriale del testo. Calcolare la cosine similarity permette di calcolare il coseno dell'angolo formato dai due vettori. In tal modo, anche se due documenti non sono simili secondo la distanza Euclidea (ad esempio, la parola "cricket" appare 50 volte nel documento 1 e 10 volte nel documento 2) essi potrebbero formare un piccolo angolo e più è piccolo l'angolo più la similarità è maggiore. La *cosine similarity* tra due testi (p, q) è calcolata nel seguente modo:

$$SIM(p,q) = \frac{\sum_{i=1}^{n} p_i \cdot q_i}{\sqrt{\sum p_i^2} \cdot \sqrt{\sum q_i^2}}$$

Per spiegare l'efficienza di tale metrica analizziamo il seguente esempio [4]: supponiamo di avere 3 documenti basati su due giocatori di cricket: *Sachin Tendulkar* e *Dhoni*. I documenti A e B sono presi da Wikipedia mentre il terzo documento (C) è un piccolo frammento del documento B. Come si può notare tutti e tre i documenti sono connessi da un tema comune: il gioco del Cricket. Il nostro obiettivo è quantificare la similarità tra i documenti. Per una facile comprensione consideriamo solo le prime 3 parole in comune tra i documenti: *Dhoni*, *Sachin* e *Cricket*.

[3] *Cosine similarity*. https://en.wikipedia.org/wiki/Cosine_similarity

[4] *Cosine Similarity - Understanding how it works*, https://www.machinelearningplus.com/nlp/cosine-similarity/

Figura 2.1: Due documenti relativi a due giocatori di cricket

Siccome il documento C è un frammento del documento B ci aspetteremmo una similarità maggiore rispetto la similarità tra il documento A e B ma se guardiamo le relative rappresentazioni (tabella 2.1) applicando la *Euclidean distance* otteniamo una similarità maggiore tra A e C, che è ciò che vogliamo evitare. Il risultato più coerente lo otteniamo utilizzando la *cosine similarity*.

	# parole comuni	*Euclidean distance*	*Cosine similarity*
Doc A *Doc B*	10 + 50 + 20 = 80	432.4	0.15
Doc A *Doc C*	10 + 5 + 1 = 16	204.02	0.23
Doc B *Doc C*	10 + 5 + 1 = 16	401.85	0.97

Se proiettiamo la rappresentazione vettoriale dei tre documenti in un spazio tridimensionale, dove ogni dimensione è la frequenza delle parole

"*Sachin*", "*Dhoni*" o "*Cricket*", otteniamo la figura 2.2:

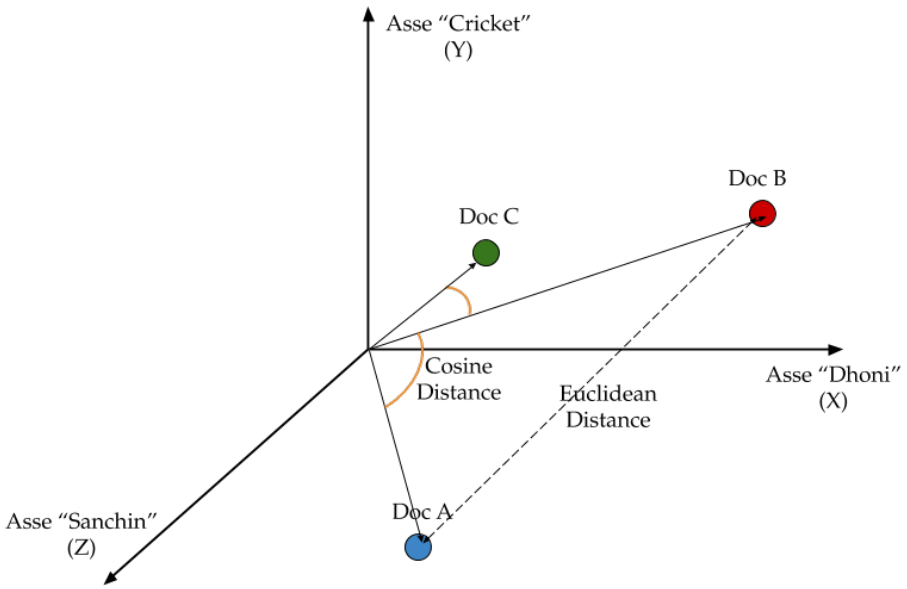


Figura 2.2: Rappresentazione in uno spazio $\mathbb{R}^3$ dei documenti A,B,C

Come si può vedere il documento dalla proiezione in figura 2.2, $Dhoni_{small}$ e il documento più grande *Dhoni* sono molto più vicini rispetto la *Euclidean distance* tra i due documenti indicando perciò una similarità più evidente con la *cosine similarity*.

Indice di Jaccard L'indice di Jaccard [5], o *Jaccard similarity*, è una tecnica basata sulle co-occorrenze delle parole. Siano S e T due insiemi di parole contenute nei documenti. La Jaccard similarity degli insiemi S e T è uguale al rapporto tra la dimensione della loro intersezione e la dimensione della loro unione.

$$J(S,T) = SIM(S,T) = \frac{|S \cap T|}{|S \cup T|}$$

Consideriamo le seguenti due frasi di cui vogliamo calcolare l'indice di Jaccard:

[5] *Jaccard index*, https://en.wikipedia.org/wiki/Jaccard_index

1. s_1 = *"AI is our friend and it has been friendly"*
2. s_2 = *"AI and humans have always been friendly"*

Per un ottenere un valore più coerente, è stata eseguita una fase di *lemmatizzazione* (riduce le parole alla stessa radice) per le due frasi considerate. Vedremo più nel dettaglio questo aspetto nei capitoli successivi. Il diagramma di *Venn* che si ottiene è mostrato nella figura 2.3.

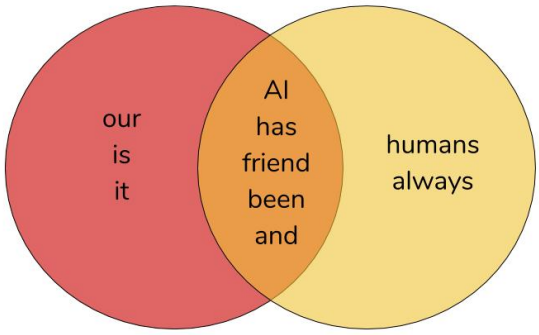


Figura 2.3: Diagramma di Venn relativo a due frasi

Quindi la *jaccard similarity* tra s_1 e s_2 è: $J(s1, s2) = 5/10 = 0.5$.

Hamming distance La *distanza di Hamming*[6] è utilizzata quando la rappresentazione vettoriale dei documenti è formata da booleani. Tale distanza tra due vettori è definita come il numero di componenti differenti tra questi. Ad esempio siano $d_1 = 10101$ e $d_2 = 11110$ due rappresentazione dei rispettivi documenti. La distanza di Hamming tra d_1 e d_2 è pari a 3: su 5 features condividono solo 2 elementi nella stessa posizione.

[6] *Hamming Distance*, https://www.tutorialspoint.com/what-is-hamming-distance

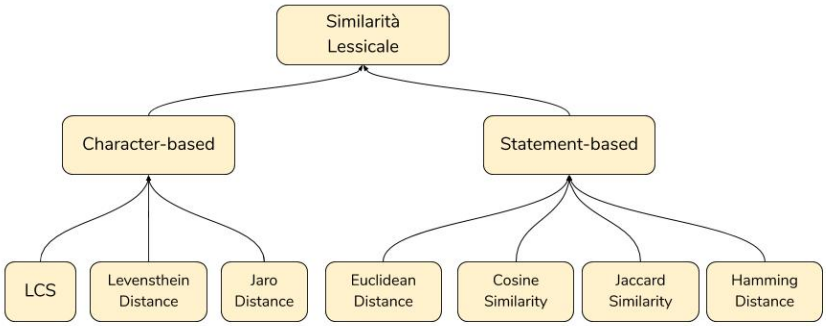


Figura 2.4: Categorizzazione metriche per la similarità lessicale

Le tecniche per il calcolo della similarità lessicale viste si possono dividere in due categorie (figura 2.4): *Character-based* e *Statement-based*.

2.3 Similarità semantica

Finora abbiamo discusso di similarità lessicale. Un'altra nozione di somiglianza per lo più studiata dalla comunità di ricerca *NLP (Natural Language Processing)* è quella relativa al quanto *semanticamente* simili siano due testi. Se osserviamo le frasi "*il gatto ha mangiato il topo*" e "*il topo ha mangiato il cibo del gatto*" sappiamo che, mentre le parole si sovrappongono in modo quasi perfetto, queste due frasi hanno in realtà un significato diverso. Ottenere il significato dei testi è spesso un compito più difficile in quanto richiede un livello più profondo di analisi. In questo esempio, possiamo effettivamente esaminare aspetti semplici come l'ordine delle parole:

$$gatto \rightarrow ha\ mangiato \rightarrow topo$$
$$\text{e}$$
$$topo \rightarrow ha\ mangiato \rightarrow cibo\ del\ gatto$$

Sebbene le parole si sovrappongano, in questo caso l'ordine influisce sul significato delle due frasi. Questo è solo un semplice esempio. L'uso dell'analisi sintattica aiuta spesso a ricercare quella semantica la quale è spesso utilizzata per affrontare problemi NLP come la generazione di risposte automatiche.

Anche qui si possono dividere le metriche in due categorie (figura 2.5): *Corpus-based* e *Knowledge-based*. Nella prima la similarità è ottenuta attraverso la conoscenza appresa dall'intera collezione di documenti (corpus) mentre nella seconda categoria la similarità è calcolata identificando il grado di similarità tra le parole e il loro uso dalle informazioni derivanti dalla rete semantica. La rete semantica più popolare è senz'altro *WordNet* ossia un database di parole in lingua inglese collegate tra di loro sulla base della loro relazione semantica.

Normalized Google Distance (NGD) La *normalized Google distance* è una metrica di similarità *corpus-based* in cui la similarità è calcolata tra le keyword sulla base del numero di suggerimenti ottenuti dalle ricerche Google. La NGD tra due keyword x e y è definita nel seguente modo:

$$NGD(x,y) = \frac{\max\{\log f(x), f(y)\} - \log f(x,y)}{\log N - \min\{\log f(x), f(y)\}}$$

dove N è il numero totale di pagine risultati dalle ricerche effettuate; *f(x)* e *f(y)* sono rispettivamente il numero di suggerimenti relativi le ricerche sui termini x e y; *f(x,y)* è il numero di pagine in cui sono presenti sia x che y.

Latent Semantic Analysis (LSA) È una delle più popolari tecniche *corpus-based*. LSA si basa sulla seguente idea: le parole che sono vicine di significato appariranno in frammenti simili di testo. Ad esempio, consideriamo le seguenti tre frasi:

1. s_1 = "*The man walked the dog*"
2. s_2 = "*The man took the dog to the park*"
3. s_3 = "*The dog went to the park*"

La tabella 2.2 contiene la frequenza delle parole per ogni frase: le righe identificano le parole mentre le colonne le frasi. Successivamente viene applicata una tecnica chiamata *Singular Value Decomposition (SVD)* il cui scopo è quello di ridurre il numero di colonne preservando la similarità strutturale

tra le righe. Infine viene utilizzata la *cosine similarity* per calcolare la similarità tra le parole ottenendo in tal modo una relazione tra di esse. LSA è spesso usata come tecnica per la riduzione della dimensionalità.

	s_1	s_2	s_3
the	2	3	2
man	1	1	0
walked	1	0	0
dog	1	1	1
took	0	1	0
to	0	1	1
park	0	1	1
went	0	0	1

Tabella 2.2: Matrice su cui viene applicata SVD

Vector similarity È una tecnica *knowledge-based* in cui viene creata una matrice per determinare la similarità tra parole. Viene creato un vettore per ogni parola appartenente alla collezione di documenti il cui valore è ottenuto dal database di *WordNet*. Ogni documento è, infine, rappresentato come un vettore che è la media dei vettori delle parole da cui è formato.

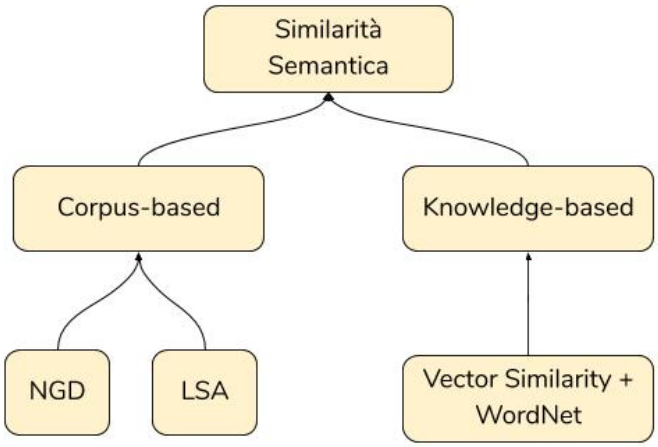


Figura 2.5: Categorizzazione metriche per la similarità semantica

2.4 Overview

Nel caso di studio di questo elaborato, siamo interessati a capire quale sia la metrica che meglio si abbina con il task. Ma la sola metrica non basta. Infatti, immaginando di aver estratto dalla collezione di documenti tutti i possibili frammenti relativi le parti interessate (3-words, frasi, paragrafi, sezioni), con gli strumenti visti fino ad ora siamo in grado di applicare solamente una ricerca lineare (*brute force*) rendendola impraticabile nel caso di una collezione formata da milioni di documenti. Necessitano quindi delle tecniche che permettano di recuperare gli oggetti più simili in maniera efficiente senza perdere di precisione. Quest'ultimo aspetto verrà trattato e spiegato nei capitoli successivi ma per il momento ci basti immaginare di aver accesso ad un motore di ricerca (*engine*) che ci fornisca quelli che sono gli elementi più simili in un tempo ragionevole. Sugli oggetti in output di tale engine, dobbiamo applicare un'operazione di *ordinamento*, anche chiamata *ranking*. Come vedremo più avanti, la metrica utilizzata in questa fase è la distanza di Levenshtein che ci permette di quantificare il numero di operazioni necessarie affinchè due stringhe siano identiche e per ottenere un valore compreso tra 0 e 1 (1 massima similarità) adotteremo la versione normalizzata.

Una volta ottenuti i candidati simili alla query in input, applicheremo ad ognuno di essi la funzione della distanza di Levenshtein normalizzata andando a scartare quelli che non soddisfano una certa soglia. Cosi facendo, con un meccanismo di ricerca efficiente e una fase di ranking, saremo in grado di fornire un sistema per il supporto testuale. La figura 2.6 mostra una visione parziale del lavoro svolto come progetto di tesi: partendo da una collezione di documenti il sistema ne elabora il contenuto e fornisce all'utente un meccanismo di recupero e confronto per l'editing di documenti fortemente strutturati. Come detto precedentemente, agli output dell'engine è applicata un'operazione di *ranking* in quanto l'utente può scegliere, eventualmente, di non voler considerare quei risultati il cui valore di similarità con la query è più basso di una determinata soglia.

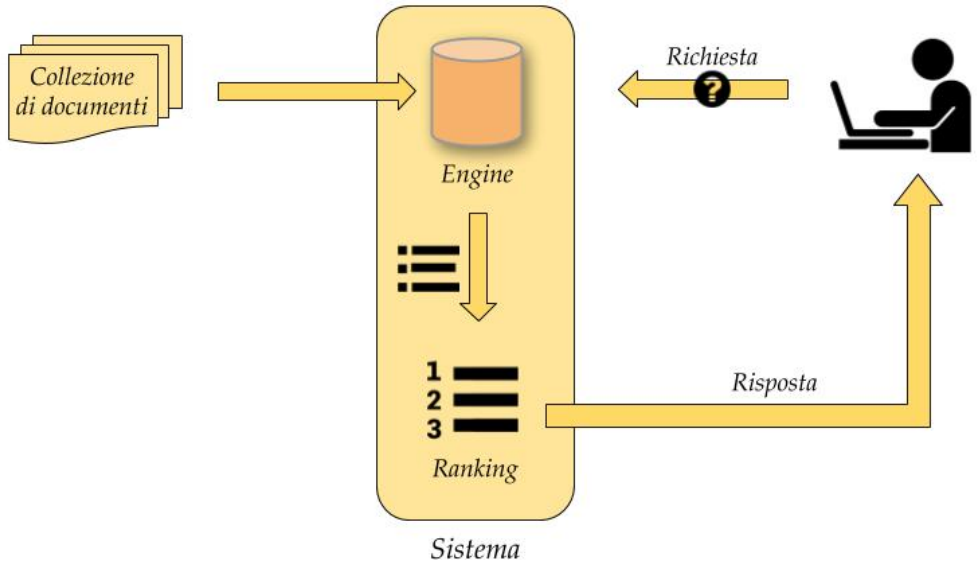


Figura 2.6: Utilizzo del sistema di supporto

Misurare la similarità tra documenti è un'importante operazione nel campo del text mining. In questo capitolo è stato analizzato il termine similarità e più nel concreto sono stati visti tre approcci per quantificare tale similarità: *String-based*, *Corpus-based* e *Knowledge-based*. La prima ha lo scopo di identificare le similarità lessicale tra due oggetti testuali e verrà fortemente utilizzata per il raggiungimento del task previsto da questa dissertazione. Corpus-Based e Knowledge-based sono approcci per quantificare la similarità semantica. In alcuni articoli scientifici questi tre approcci sono combinati assieme per formare delle tecniche di misurazione ibride.

Nel capitolo successivo andremo a discutere alcune delle tecniche più note in letteratura per ottenere una rappresentazione vettoriale (firma o *signature*) partendo da oggetti di natura testuale tenendo in considerazione l'importante proprietà di preservare la similarità anche dopo tale trasformazione. Quindi se due documenti hanno un certo valore di similarità (identificato da una delle metriche viste nelle capitolo corrente) ci aspetteremmo che tale quantificata relazione venga preservata anche dopo le rispettive generazione delle firme.

Capitolo 3

Text Representation

Nel capitolo precedente sono stati discussi concetti che ci serviranno nella dimostrazione della tesi e più nello specifico abbiamo dato una definizione sia concettuale che matematica del termine similarità. Inoltre, sono state analizzate due diverse forme di similarità, lessicale e semantica. Concentrandoci maggiormente sull'analisi della prima, sono state fornite tecniche che permettono di quantificare la somiglianza tra due testi. La conclusione del capitolo precedente ha descritto come l'utilizzo della sola metrica non basti a creare una sistema efficiente e che quindi si necessitano di tecniche di ricerca che possano rapidamente identificare gli elementi più simili in una collezione molto ampia di documenti. Per poter permettere ciò, si devono utilizzare dei metodi che riducano la dimensione degli elementi appartenenti alla collezione attraverso una rappresentazione vettoriale di quest'ultimi. Ottenere una rappresentazione ci permette di diminuire la complessità computazione poiché la fase di confronto viene eseguita sulle relative rappresentazioni che avranno, appunto, una dimensione molto inferiore rispetto ai rispettivi documenti di partenza. Vedremo, inoltre, come poter utilizzare queste a nostro vantaggio per diminuire ulteriormente il tempo di calcolo per la ricerca di oggetti simili.

Gli algoritmi di machine learning operano in un spazio numerico di *features* (caratteristiche), aspettandosi come input array a due dimensioni in cui le righe rappresentano le istanze (i documenti) e le colonne rappresentano le features. Per applicare tali algoritmi sul testo abbiamo bisogno di

trasformare i documenti in una rappresentazione vettoriale che il calcolatore può comprendere per poi quindi applicare operazioni. Questo processo è chiamato estrazione delle features o più semplicemente ***vectorization***.

Nel campo della *text analysis*, gli input sono interi documenti di varia lunghezza appartenenti ad una qualche collezione le cui rappresentazione vettoriali però hanno lunghezza uniforme. Ogni elemento, o entry, del vettore è una feature. Per il testo, le features rappresentano gli attributi e le proprietà del documento che descrivono uno spazio multidimensionale in cui applicare algoritmi. I punti nello spazio possono essere vicini o molto lontani, strettamente raggruppati o uniformemente distribuiti. Lo spazio è quindi mappato in modo tale che documenti simili siano più vicini e quelli che sono diversi siano più distanti.

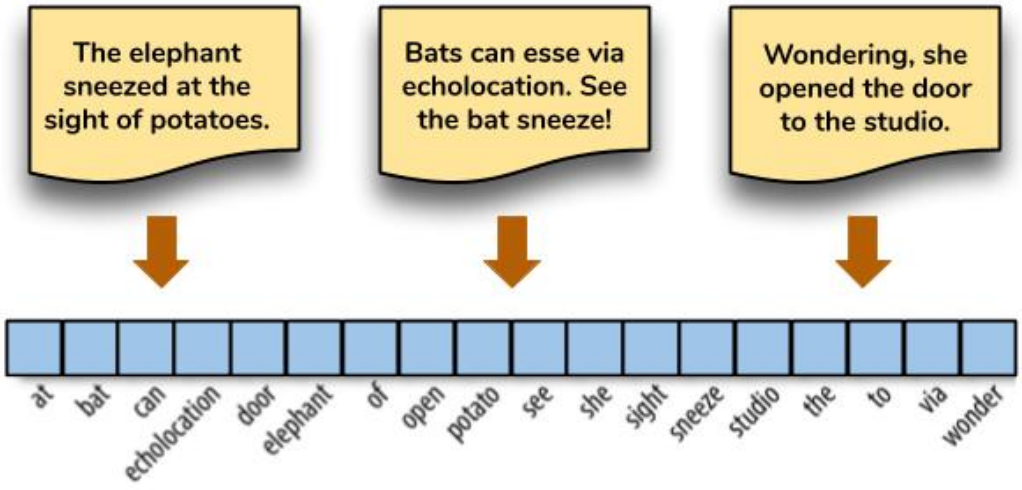
La più semplice vectorization è il modello *bag-of-word* [1], la cui intuizione è la seguente: il significato e la somiglianza tra i testi sono codificati nel vocabolario. Per esempio, le pagine di Wikipedia relative al *Calcio* e *Leo Messi* saranno probabilmente simili, anzi si può dire che alcune parole appariranno in entrambi i documenti relativi a questi due temi, mentre avranno un numero di parole condivise molto inferiori su temi che hanno a che fare con la *Brexit* o *Donald Trump*. Tale modello, per quanto semplice, è estremamente efficiente e pone le basi per modelli molto più complessi. Prima di addentraci con la descrizione dei modelli è giusto precisare che è necessario effettuare una fase di tokenizzazione e normalizzazione del testo prima di applicare le tecniche di codifica. Questi aspetti saranno spiegati nel capitolo 5 in cui verranno descritte sia le operazioni di normalizzazione e sia il dataset su cui sono applicate.

3.1 Bag-of-Words

L'idea dietro questo approccio è semplice, sebbene molto potente. Per vettorializzare un documento secondo il modello bag-of-words, dobbiamo definire un vettore a lunghezza fissa dove ogni *entry* corrisponde ad un token dell'insieme di parole, chiamato dizionario, del corpus di partenza. La dimensione

[1] *Bag-of-words model*, https://en.wikipedia.org/wiki/Bag-of-words_model

del vettore è uguale alla cardinalità dell'insieme dizionario. A scopo esemplificativo, possiamo pensare al vettore in modo che le posizione delle entry siano in ordine alfabetico.



A seconda del encoding model scelto il valore di ogni entry può variare perciò nelle successive sottosezioni verranno presentate alcune delle tecniche più famose per la rappresentazione vettoriale di un testo secondo l'approccio BOW. Prima di continuare è bene far notare che molto spesso il testo di ogni documento necessita di una fase di normalizzazione. Nell'esempio sono stati rimossi tutti i caratteri relativi la punteggiatura. Nei capitoli successivi verranno elencate ulteriori operazioni che aiutano la fase di normalizzazione del testo favorendo una ricerca più efficiente e precisa. Il modello BoW presenta, come vedremom un grande svantaggio dovuto alla *curse of dimensionality* [SLY18]: ogni vettore risultate avrà dimensione pari alla cardinalità del insieme vocabolario.

3.1.1 Count Vector

L'encoding model più semplice consiste nel riempire il vettore con la frequenza di ogni parola presente nel documento. In questo schema di codifica, ogni documento è rappresentato come una lista di token che lo compongono e il valore di ogni entry nel vettore è il suo conteggio. Questa rappresentazione

può essere sia una codifica a conteggio diretto (intero), come mostrato nella nella figura 3.1, sia una codifica normalizzata in cui ogni parola è ponderata dal numero totale di parole nel documento.

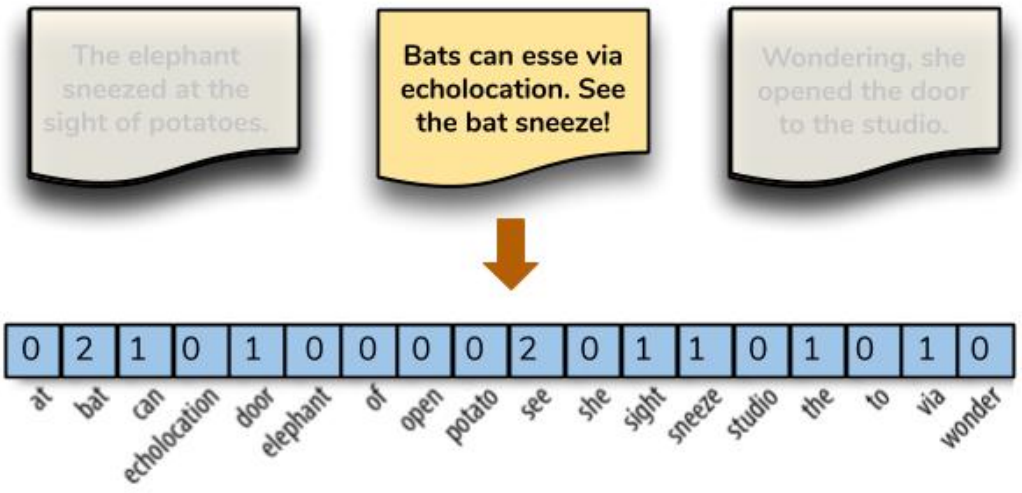


Figura 3.1: Rappresentazione con modello *count vector*

3.1.2 One-Hot Encoding

Il metodo *count vector* non tiene conto della grammatica e della posizione delle parole nei documenti ed inoltre soffre della *distribuzione Zipf* [Zip49]: "*l'80% delle volte vengono utilizzate il 20% delle parole*". Il risultato è che token che appaiono frequentemente sono ordini di grandezza più "significativi" di altri. Questo può avere un significativo impatto su alcuni modelli. Una soluzione a questo problema è la codifica *one-hot*, ossia un metodo che utilizza un vettore booleano per indicare la presenza (1) o assenza (0) di un token all'interno di un documento.

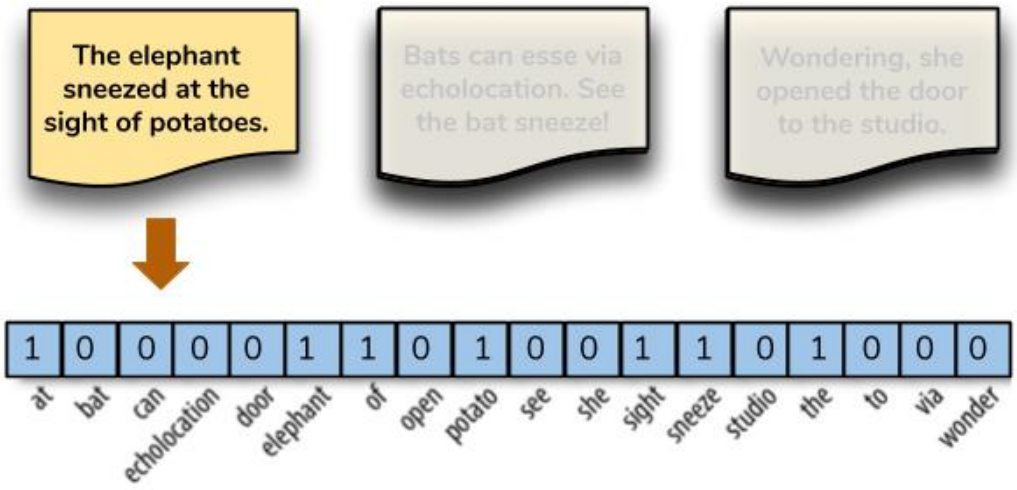


Figura 3.2: Rappresentazione con modello *one-hot*

La codifica *one-hot* riduce in tal modo il problema dello sbilanciamento della distribuzione dei tokens, semplificando la rappresentazione di un documento. Questa riduzione è molto efficace per documenti di piccola dimensione (come i tweets) che non contengono molti termini ripetuti. Inoltre, questa rappresentazione permette di indicare la similarità in base all'intero documento e non sulla singola parola. Questo perché tutte le parole sono equidistanti rendendo impossibile quantificare la loro similarità. Come normalizzare i tokens, attraverso *stemming* o *lemmatization*, è un argomento che verrà affrontato nei capitoli successivi.

3.1.3 N-Grams

Un altro tipo di tecnica tiene conto di insiemi di parole adiacenti. Il metodo *N-Grams* non si focalizza sulle singole parole ma prende in considerazione sequenze contigue di n token. Ad esempio, con la frase "*il gatto ha mangiato il topo*", utilizzando $n = 2$ otteniamo il seguente insieme di *2-grammi*:

$$\{(il,\ gatto),(gatto,\ ha),(ha,\ mangiato),(mangiato,il),\ (il,topo)\}.$$

Un modello di questo genere permette di catturare la probabilità congiunta che una sequenza di parole appaia all'interno del testo considerato.

3.2 Term Frequency–Inverse Document Frequency

Le codifiche *bag-of-word* che abbiamo visto finora descrivono solo un documento in modo indipendente, senza tenere in considerazione il contesto dell'intera collezione di documenti in gioco. Un approccio migliore sarebbe quello di considerare la frequenza relativa o la rarità dei token nel documento rispetto alla loro frequenza in altri documenti. Secondo la codifica *TF-IDF* [Jon72] le parole comuni presenti in tutti i documenti sono quelle meno importanti è quindi preferibile concentrare l'attenzione sulle parole specifiche che descrivono al meglio il tema di un documento. Ad esempio, in un corpus di testi economici, token come "*spread*", "*PIL*" e "*banca*" appariranno più frequentemente rispetto a documenti che parlano di sport, mentre altri token possono essere condivisi in altri temi in quanto parole di uso comune e quindi meno importanti. L'encoding basato su *TF-IDF* accentua i termini più rilevanti per un documento specifico, come mostrato nell'esempio di figura 3.3, dove il token "*studio*" ha un'alta rilevanza in questo documento dal momento che appare solo in quel testo (il valore nelle celle non è quello reale ma serve solo per avere un'idea iniziale del funzionamento).

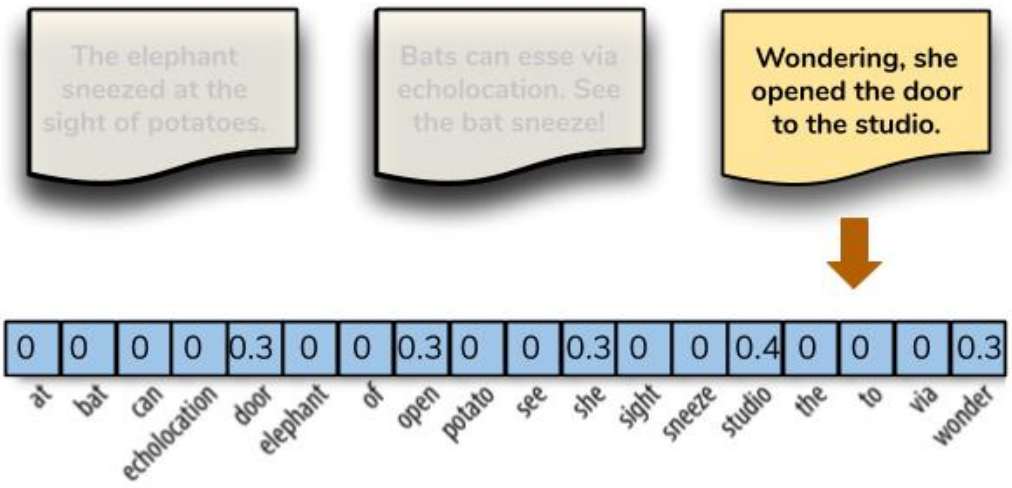


Figura 3.3: Rappresentazione con modello *tf-idf*

La funzione *TF-IDF* può essere scomposta in due fattori:

- *TF (Term Frequency)* dove $n_{i,j}$ è il numero di occorrenze del termine i nel documento j, mentre $|d_j|$ è il numero di parole nel documento j

$$tf_{i,j} = \frac{n_{i,j}}{|d_j|}$$

- *IDF (Inverse Term Frequency)*, indica l'importanza generale del termine i nella collezione dove $|D|$ è il numero di documenti nel corpus e il denominatore è il numero di documenti che contengono quel token i

$$idf_i = \log \frac{|D|}{|\{d : i \in D\}|}$$

Perciò la *TF-IDF* del termine i rispetto al documento j sarà pari a:

$$(tf - idf)_{i,j} = tf_{i,j} \times idf_i$$

Per avere una comprensione più chiara consideriamo due frasi ($|D| = 2$) :

1. s_1 = "*The car is driven on the road*"
2. s_2 = "*The truck is driven on the highway*"

Calcoliamo ora il valore *TF-IDF* per le frasi precedenti (l'insieme delle frasi rappresenta il corpus).

token	TF		IDF	$TF \times IDF$	
	s_1	s_2		s_1	s_2
The	1/7	1/7	$\log(2/2) = 0$	0	0
Car	1/7	0	$\log(2/1) = 0.3$	0.043	0
Truck	0	1/7	$\log(2/1) = 0.3$	0	0.043
Is	1/7	1/7	$\log(2/2) = 0$	0	0
Driven	1/7	1/7	$\log(2/2) = 0$	0	0
On	1/7	1/7	$\log(2/2) = 0$	0	0
Road	1/7	0	$\log(2/1) = 0.3$	0.043	0
Highway	0	1/7	$\log(2/1) = 0.3$	0	0.043

Tabella 3.1: Tabella con i valori *TF-IDF* relativi l'esempio

Dalla tabella 3.1 possiamo vedere come il valore *TF-IDF* delle parole comuni è pari a 0. Mentre le parole più rilevanti sono quelle che hanno valore *TF-IDF* maggiore di 0 e quindi "*car*" , "*truck*", "*road*" e "*highway*". Uno dei benefici di *TF–IDF* è quello che risolve il problema dei termini più frequenti (come gli articoli) in maniera del tutto naturale. Questi termini sono detti *stopwords* e lo score relativo a questi avranno un valore TF-IDF molto basso.

Nel capitolo corrente abbiamo visto alcune metodologie che ci permettono di poter rappresentare un testo attraverso una rappresentazione vettoriale. Una rappresentazione di questo genere può facilitare l'adozione di algoritmi che permettono di lavorare in uno spazio a più dimensioni in cui ognuna di queste rappresenta una proprietà (*feature*) del documento. Nella dissertazione corrente sarà molto importante trovare una giusta rappresentazione in quanto è tramite questa che applicheremo una delle tecniche più conosciute in letteratura per la risoluzione del cosiddetto *Nearest Neighbor problem*. Vedremo come questo problema può essere formalizzato per raggiungere l'obiettivo delle tesi: trovare elementi simili sulla base di data una metrica di similiarità.

Capitolo 4

Nearest Neighbors Problem

Nel capitolo precedente sono state analizzate alcune delle tecniche più conosciute per la rappresentazione in forma vettoriale di un testo. L'aspetto più importante emerso dalle tecniche viste è stato quello di generare una rappresentazione che potesse preservare la similarità lessicale tra due documenti. Quindi dati due documenti con una certa similarità ci aspetteremmo che la relazione di similarità che intercorre tra questi si rifletta anche sulle relative rappresentazioni. Come vedremo in questa parte, per ottenere un risultato in tempi brevi dobbiamo accontentarci di metodi che ci permettono di ottenere una stima della similarità.

In questo capitolo andremo a discutere del *nearest neighbors problem* e di come le relative soluzioni possano aiutare al raggiungimento della tesi di questa dissertazione. Il capitolo inizia con l'introduzione al problema e di alcuni campi di applicazione; successivamente verranno presentate due tecniche la cui implementazione verrà descritta nel capitolo 5.

4.1 Cos'è il Nearest Neighbors Problem?

Il compito principale di un motore di ricerca... è la ricerca! Uno dei fattori che rende un motore di ricerca veloce e preciso è quello di avere un indice libero da duplicati [MK19]. Per rimuovere i duplicati o duplicati quasi perfetti, *duplicate and near-duplicate (DND)*, un motore di ricerca ha bisogno di un *document detection system* testuale che possa identificare le coppie di

duplicati. Un search engine lavora velocemente e in maniera affidabile quando il suo indice ha le informazioni sufficienti e necessarie. Per questo motivo, i documenti DND dovrebbero essere rimossi dal suo indice.
Un altro componente importante e tipico dei motori di ricerca (come Google) soggetto ad un buon document detection system è il *crawler*. Il crawler è un componente che ha il compito di scaricare le pagine web per poi indicizzarle e quindi affronta il problema dei DND dal momento che nel web esistono molte pagine di questo tipo.

Per avere una definizione formale degli DND possiamo affermare che le pagine *duplicates* sono quelle identiche in termini di contenuto e che sono accessibili attraverso vari URLs. Invece, i *near-duplicates* sono quelle pagine con delle leggere differenze, come un cambio di data o qualche altra modifica di minore entità, che rendono le due pagine non identiche. Uno dei problemi con i DND è che la loro presenza aumenta la dimensione dell'indice di ricerca. Tradizionali approcci al problema, come ad esempio un ricerca *brute-force* o l'utilizzo di un semplice algoritmo di hash, non sono adatti per costruire un efficiente sistema di rilevazione. Gli approcci basati su un a ricerca brute force confrontano ogni documento con tutti gli altri usando una delle metriche viste nel capitolo 2, come la *cosine similarity* o l'*indice di jaccard*. Questi approcci non sono adatti per collezioni di documenti molto grandi che contengono centinaia di migliaia di documenti oltre a non essere efficienti dal momento che tutti i documenti devono essere comparati uno ad uno. La complessità per confrontare tutti i documenti è pari a $O(n^2 * L)$ dove L è la lunghezza media dei documenti e n sono il numero di documenti della collezione.

Una delle soluzioni più utilizzate in letteratura è quella di ridurre la dimensione di L usando metodi basati sul calcolo di funzioni hash. Questi metodi generano una firma (molto più piccola della dimensione del documento a cui è applicato) per ogni documento. Ad esempio, si può generare una firma di lunghezza pari a 128 o 160 bit utilizzando funzioni hash come *MD5* o *SHA1*. Così facendo, il tempo necessario per confrontare due documenti è drasticamente ridotto dal momento che si andrà a confrontare la rispettiva firma e non ci sarà più un confronto tra le parole dei documenti. Lo svantaggio di questa metodologia è la totale mancanza nel rilevare i *near-*

duplicates in quanto una piccola differenza tra due input produrrà una firma totalmente diversa. È necessario quindi individuare un approccio differente per il rilevamento degli DND.

Il *near duplicate detection problem* è stato ben studiato e sono state proposte una varietà di soluzioni. Una di queste è stata proposta da Broder [Bro97] e prevede una tecnica per il calcolo della somiglianza, *resemblance*, di due documenti, ognuno dei quali è diviso in frammenti chiamati *shingles*. Gli shingles non dipendono da alcuna conoscenza linguistica ma semplicemente da una fase di tokenizzazione del documento in una lista di parole. Se due documenti condividono una certa percentuale di shingles possono essere considerati simili e marcati come near-duplicates.

Un altro approccio è quello che deriva dal problema dei ***Nearest Neighbor*** *(NN)* definito nel seguente modo: *Dato un insieme* $P = \{ p_i \mid p_i \in \mathbb{R}^d, 1 \leq i \leq n\}$ *e una funzione* D *che definisce la distanza tra i punti, l'obiettivo è costruire una struttara dati che dato un nuovo punto* $q \in \mathbb{R}^d$ *produca un* $NN(q) = arg\min_{p \in P} D(p, q)$.

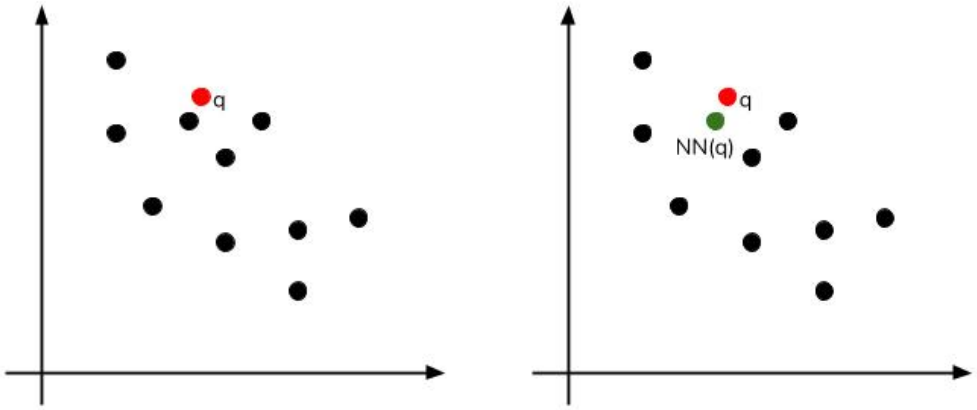


Figura 4.1: Esempio di un problema NN.

La figura 4.1 mostra un esempio di un'istanza del problema Nearest Neighbor. Sulla sinistra è presente un insieme P di 10 punti (N=10) in uno spazio bi-dimensionale con un query point q. La parte destra mostra la soluzione del problema NN(q). Possiamo quindi affermare che il problema

appena descritto può essere visto come un problema di ottimizzazione in cui la funzione obiettivo da minimizzare è la distanza D con il punto q.

Esiste anche una variante, chiamata ***R-near neighbor***, che permette di ottenere l'insieme dei punti più vicini tenendo in considerazione un vincolo sulla distanza. L'*R-NN problem* è definito nel seguente modo: *Dato un insieme* $P = \{ p_i \mid p_i \in \mathbb{R}^d, 1 \leq i \leq n\}$, *una funzione* D *che definisce la distanza tra i punti e una valore limite* R *per relativo alla distanza considerata, l'obiettivo è costruire una struttara dati che dato un nuovo punto* $q \in \mathbb{R}^d$ *produca un'insieme* $R - NN(q) = \{p \,|\, dist(p, q) \leq R, p \in P\}$.

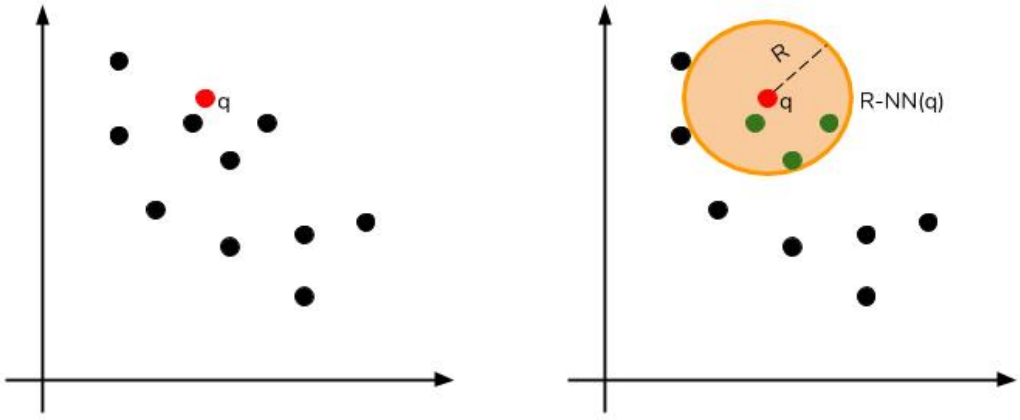


Figura 4.2: Esempio di un problema R-NN.

La figura 4.2 mostra un esempio di un'istanza del problema R-Near Neighbor. Sulla sinistra è presente un insieme P di 10 punti (N=10) in uno spazio bi-dimensionale con un query point q. La parte destra mostra la soluzione al problema: tutti i punti all'interno dell'area soddisfano il vincolo sulla distanza R.

Naturalmente, i problemi *NN* e *R-NN* sono collegati. È facile vedere come il *Nearest Neighbor problem* risolva anche *R-Near Neighbor problem* - basta semplicemente controllare se il punto risultante p sia un R-near neighbor ossia se $||p - q|| \leq R$. A questo punto il nostro obiettivo si riduce a trovare una buona rappresentazione da applicare all'intera collezione di documenti su cui si vuole fare la ricerca e successivamente utilizzare le tecniche per il

problema sopra citato per recuperare quelle che sono le possibili coppie di near-duplicates.

4.2 Applicazioni del Nearest Neighbors Problem?

Prima di analizzare le possibili implementazioni per la risoluzione del problema è doveroso elencare alcuni campi in cui il problema sopra citato si presenta. Nel capitolo 2 abbiamo focalizzato la nostra attenzione su una particolare nozione di similarità, quella lessicale, e abbiamo anche visto un tipo di metrica che ci permette di quantificarla attraverso la rappresentazione del documento in un insieme. Questo tipo di metrica è la *Jaccard similarity*. Per trasformare il documento in un insieme, useremo la tecnica chiamata "*shingling*". Tale metrica affronta bene il problema della similarità testuale quindi è necessario tenere a mente che ciò che esamineremo è la similarità a livello carattere. La similarità lessicale ha molti utilizzi e sicuramente la *detection* degli DND è uno di quelli.

Nella maggior parte delle applicazioni, come quelle che stiamo per descrivere, i documenti che cerchiamo non sono identici ma condividono una grande porzione di testo.

Plagiarism Detection L'autore del plagio potrebbe estrarre solo alcune parti di un documento alterando qualche parola e l'ordine con cui le frasi compaiono nel documento originale. Ciò che ne deriva è un documento che potrebbe contenere circa il 60% o più dell'originale. Non è un semplice processo quello di confrontare documenti carattere per carattere e non è la soluzione più efficiente per la creazione di un plagiarism detector.

Mirror pages Come già accennato nella sezione precedente, un campo in cui si necessita di risolvere il problema dei duplicati è quello del Web. Capita spesso, per siti Web popolari o importanti, di duplicare il numero copie del sito in cui l'URL è differente. Pagine di questo tipo sono chiamate *mirror* e sono abbastanza simili, ma raramente identiche. Ad esempio, pagine relative

ai corsi universitari potrebbero includere degli avvisi, compiti e materiale didattico. Pagine simili potrebbero cambiare il nome del corso, anno e qualche altre informazione di contorno. È importante essere in grado di determinare pagine come questo tipo in modo da produrre risultati migliori senza dover mostrare due pagine che sono quasi identiche.

Un'altra classe di applicazione dove la similarità di insiemi è molto importante è chiamata *collaborative filtering*: processo in cui viene consigliato all'utente oggetti apprezzati da altri utenti con cui si ha un qualche collegamento.

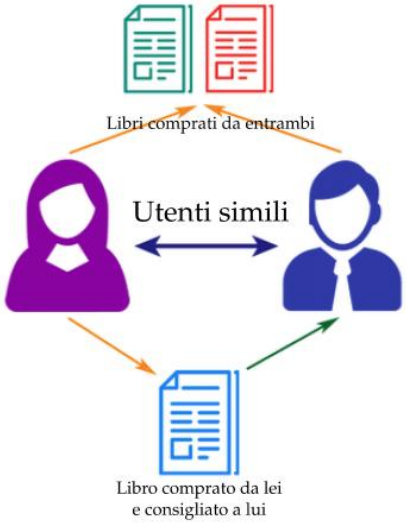


Figura 4.3: Funzionamento *collaborative filtering*

Vendite Online *Amazon* ha milioni di utenti e vendite di oggetti ogni anno. Il suo database registra quali oggetti sono stati comprati e da quale utente. Possiamo affermare che due utenti sono simili (hanno gusti simili) se il loro insieme di acquisti ha un'alta similarità di jaccard. Notiamo che, mentre ci aspetteremmo di avere una similarità pari o superiore al 90% per le mirror pages dette prima, in questo caso anche una similarità di jaccard pari al 20% può bastare per identificare due utenti con gusti simili.

Movie rating Anche nel campo delle piattaforme streaming è possibile applicare lo stesso concetto utilizzato per le vendite. Netflix, ad esempio, registra i film che un certo utente visiona e anche quale voto assegna. Possiamo pensare ai film come oggetti simili se questi sono stati visti e votati dagli stessi utenti e successivamente identificare sulla base di queste operazioni quali sono gli utenti che più si assomigliano. La stessa osservazione fatta nel caso delle vendite Amazon può essere applicata in questa situazione: non si necessita di una alto grado di similarità, ma semplicemente una relazione, anche piccola, che possa accomunare i gusti degli utenti. Quando il dataset è costituito da dati numerici, come uno score, piuttosto che da decisioni binarie (lo compro/non lo compro oppure mi piace/non mi piace), non possiamo rappresentare tale dataset come un semplice insieme di utenti o oggetti. Alcune opzioni da considerare sono:

1. Ignorare i film in cui l'utente ha assegnato un voto basso, così facendo, trattiamo questi eventi come se l'utente non abbia mai visto quel film;

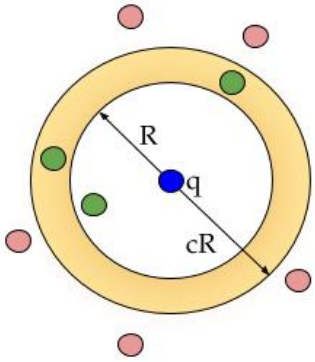
2. Quando si confrontano due utenti, immaginiamo due insiemi per ogni film: "mi piace" e "non mi piace". Se un utente ha assegnato un voto alto ad un film, questo viene inserito nell'insieme "mi piace", altrimenti nell'insieme "non mi piace". Successivamente, possiamo calcolare la similarità di jaccard tra questi due insiemi degli utenti desiderati e lo stesso meccanismo può essere applicato ai film;

3. Se le valutazioni sono espresse in una scala da 1 a 5 stelle, possiamo inserire il film nell'insieme dell'utente n volte se le stelle assegnate al film sono n. Viene utilizzata, poi, la *Jaccard similarity for bags* per misurare la similarità tra due utenti. La similarità per le borse (bags) A e B è definita contando un elemento n volte nell'intersezione sole se n è il numero minimo di volte che l'elemento appare in A e B. Nell'unione, si conta la somma degli elementi contenuti in A e in B. Ad esempio, consideriamo due insiemi per gli utenti A e B, immaginando che ogni elemento distinto identifichi un film e che il numero di occorrenze di quell'elemento rappresenti il voto dato a quel film: A = {a,a,a,b}, B = {a,a,b,b,c}. La *bag-similarity* è 1/3. L'intersezione conta due volte a

e una volta b. La dimensione dell'unione tra i due insiemi è sempre la somma della delle cardinalità dei due insiemi (9 in questo caso).

4.3 Locality Sensitive Hashing

Il problema della *similarity search*, anche conosciuto come *nearest neighbor search*, è quello di trovare il punto più vicino rispetto a un determinato oggetto in input (una query) e in base ad una qualche metrica di similarità. Nel caso in cuila collezione di oggetti abbia dimensioni molto grandi o il calcolo della distanza tra la query e gli oggetti sia molto costosta, è spesso computazionalmente impraticabile trovare il risultato esatto (*exact nearest neighbor*). Sono presenti molti algoritmi efficienti in letteratura che permettono di lavorare in uno spazio di piccole dimensioni (da 10 a 20). Nel 1975 fu introdotta da Jon Bentley una struttura dati chiamata *kd-tree* [Ben75]. La soluzione proposta da Bentley però soffre sia in spazio che in tempo che è esponenziale nel numero delle dimensioni (curse of dimensionality).
Per trovare un *NN(q)* si potrebbero salvare tutti i punti $p \in P$ e applicare una ricerca lineare andando a calcolare il valore $D(q,p)\ \forall p \in P$ e prendere il punto p con valore della funzione D minimo. Questo approccio è ovviamente impraticabile nel caso di una collezione molto grande di punti. La ricerca di soluzioni efficienti per il *nearest neighbor problem* ha portato a considerare se un possibile rilassamento del problema potesse migliorare le prestazione in tempo e spazio. Negli ultimi anni molti ricercatori hanno proposto un approccio differente per risolvere il problema della dimensionalità attraverso il problema dell'***Approximate Nearest Neighbor Search*** [AIR18] la cui definizione è la seguente: *Dato un insieme* $P = \{\ p_i \mid p_i \in \mathbb{R}^d, 1 \leq i \leq n\}$*, una funzione* D *che definisce la distanza tra i punti, una distanza* R*, l'obiettivo è costruire una struttura dati che dato un nuovo punto* $q \in \mathbb{R}^d$ *produca un insieme* $(c,R) - NN(q) = \{\ p' \mid dist(p',q) \leq c \cdot \min_{p \in P} dist(q,p) \wedge dist(q,p) \leq R\}$.



Per ridurre il tempo necessario per la ricerca di un esatto nearest neighbor e la quantità di memoria utilizzata, è stata proposta la ricerca per approssimazione e negli articoli [AMN+98] [Kle97] sono presenti esperimenti che mostrano che per valori di $c > 1$, con c detto fattore di approssimazione, si ottengono strutture dati efficienti ottenendo un ricerca sublineare. L'aspetto più intrigante di questo approccio è che, in molti casi, un *approximate nearest neighbor* è meglio di un singolo, ma esatto, risultato. Tuttavia, un efficiente algoritmo di approssimazione può essere usato per risolvere il *nearest neighbor problem* numerando tutti i risultati ottenuti prendendo quello più vicino.

Molte soluzioni all'*ANN problem* prevedono l'utilizzo di algoritmi di hashing che forniscono un meccanismo per ridurre la dimensionalità mappando un oggetto di dimensione M in uno di dimensione N (N << M) favorendo cosi una drastica riduzione del tempo necessario per il calcolo di similarità. Gli oggetti generati sono chiamati *hash codes*, o valori hash. Formalmente una funzione hash è definita come: $y = h(x)$, dove y è l'*hash code* e $h(\cdot)$ è la funzione. Nell'*ANN search* vengono applicate diverse funzioni hash per calcolare l'hash code: $y = h(x)$ in cui $y = [y_1, y_2, \cdots, y_M]^T$ e $[h_1(x), h_2(x), \cdots, h_M(x)]^T$, le relative funzione hash scelte. Esistono due strategie che permettono di risolvere *ANN problem* attraverso l'utilizzo degli hash code: *fast distance approximation* e *hash table lookup*.

Fast distance approximation Il modo diretto è quello di eseguire una ricerca esaustiva: l'hash code della query viene confrontato con gli altri hash

code relative agli elementi della collezione. Ogni valore relativo il confronto viene salvato in un array che viene infine ordinato per individuare gli elementi con valore di similarità maggiore. Questa strategia sfrutta due vantaggi degli hash codes: il primo è che la distanza può essere calcolata con un costo molto più basso rispetto alla distanza calcolata con l'input originale; il secondo vantaggio è che la dimensione degli hash code è molto inferiore rispetto agli input iniziali e quindi tutto può essere caricato in memoria.

Hash table lookup L'*hash table* è una struttura dati composta da bucket (celle di memoria) ognuno dei quali è indicizzato da un *hash code*. Ogni oggetto x è inserito in un bucket $h(x)$. A differenza dei tradizionali algoritmi di hashing che evitano le collisioni (due oggetti mappati nello stesso bucket), l'approccio con hashing mira a massimizzare la probabilità di collisioni per due oggetti simili. Dato una query q, gli oggetti che giacciono nel bucket $h(q)$ sono considerati come elementi simili a q. Per migliorare la recall, vengono costruite L hash tables, e gli oggetti verranno mappati in un bucket per ognuna di queste L tabelle. Per garantire, invece, una buona precisione, ogni hash code y_i ($y = [y_i]_{i=1}^{L}$) necessita essere abbastanza grande da poter permettere un numero molto alto di bucket.

Un modo pratico per accelerare il processo di ricerca è quello di eseguire una ricerca non esaustiva: prima si recupera un insieme di candidati utilizzando gli *inverted index* e successivamente si calcola la distanza tra la query e i candidati trovati. La prima strategia descritta (*hash table lookup*) è alla base di una delle tecniche più conosciute: *locality sensitive hashing (LSH)*. Il termine *locality sensitive hashing* è stato introdotto per la prima volta nel 1998 [IM98] per indicare uno schema per la ricerca di *approximate nearest neighbor* in spazi di grandi dimensioni.

LSH è basato sulla semplice idea che se due punti sono vicini allora lo saranno anche dopo un'operazione di proiezione. Ciò può essere facilmente compreso attraverso la metafora della sfera [SM08] come mostra la figura 4.4: due punti vicini sulla sfera lo saranno anche in una successiva proiezione della sfera in uno spazio bidimensionale. Al contrario, due punti lontani sulla sfera possono apparire vicini in seguito ad una proiezione in uno spazio bidimen-

sionale. Tuttavia, nella maggior parte delle proiezioni tali punti risulteranno distanti.

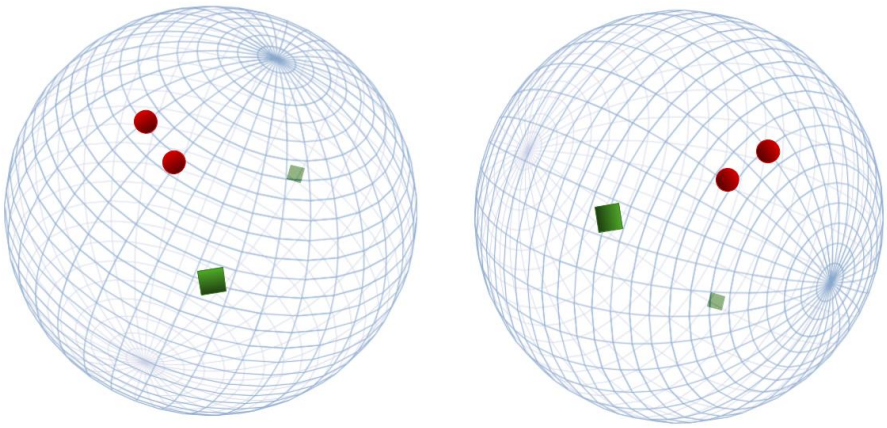


Figura 4.4: Metafora della sfera

Così come si è passati da $\mathbb{R}^3$ a $\mathbb{R}^2$, così si dovrà passare da uno spazio di grandi dimensioni a uno di piccole dimensioni cercando di preservare le proprietà di similarità degli oggetti coinvolti. Gli algoritmi basati su LSH, come vedremo, permetteranno di generare delle firme che preservano la similarità. Possiamo quindi definire LSH come una famiglia funzioni con la proprietà che oggetti simili nel dominio di queste funzioni hanno una maggiore probabilità di essere raggruppati in uno stesso bucket rispetto ad oggetti con non sono simili.

Locality-sensitive hashing Una famiglia $\mathbb{H}$ è chiamata $(R,\ cR, P_1, P_2) -$ *sensitive* se per ogni coppia di punti $p,\ q \in \mathbb{R}^d$ vale che

- se $dist(p, q) \leq R$ allora $Pr_{\mathbb{H}}[\mathrm{h(q)} = \mathrm{h(p)}] \geq P_1$,
- se $dist(p, q) \geq cR$ allora $Pr_{\mathbb{H}}[\mathrm{h(q)} = \mathrm{h(p)}] \leq P_2$

Affinché $\mathbb{H}$ sia una buona famiglia di *locality-sensitive functions* deve valere che $P_1 > P_2$. A differenza dei tradizionali algoritmi di hashing, LSH massimizza la probabilità di due oggetti simili di essere collocati nello stesso bucket.

LSH con Hamming distance Per spiegare meglio il concetto, analizziamo il seguente esempio: assumiamo che ci siano punti binari, ossia le coordinate di tali punti possono essere 0 o 1, e che la distanza tra i punti p e q sia calcolata attraverso la distanza di Hamming. In tal caso, usiamo una famiglia $\mathbb{H}$ di funzioni che prendono un vettore in input (le coordinate del punto) e restituiscono un casuale valore relativo all'i-esima coordinata dell'input. Ad esempio, se $h_i \in \mathbb{H}$ allora $h_i(p) = p_i$. Perciò scegliere una funzione hash casuale da $\mathbb{H}$ significa che $h(p)$ produce una coordinata casuale di p. Per vedere come $\mathbb{H}$ sia una famiglia di *locality-sensitive functions* osserviamo che la $Pr[h(q) = h(p)]$ è uguale al grado di similarità tra i punti p e q. Pertanto, $P_1 = 1 - R/d$, $P_2 = 1 - cR/d$ e fino a quando il fattore di approssimazione soddisfa $c > 1$ allora abbiamo che $P_1 > P_2$.

4.3.1 Minhash

Min-Wise Independent Permutations, anche conosciuto come *Minhash*, è un esempio di una famiglia di *locality-sensitive functions* che assieme alla tecnica del *banding* permette di sviluppare un approccio basato su LSH. Tale algoritmo, sviluppato da Andrei Broder nel 1997 [Bro97], permette di stimare la similarità tra insiemi utilizzando come metrica di riferimento la similarità di Jaccard. Dopo poco più di due decadi, Minhash rimane uno dei metodi basati su LSH più utilizzati e citati in letteratura.

Shingling di documenti Per poter spiegare il funzionamento di Minhash dobbiamo innanzitutto definire un modo per trasformare un documento in un insieme con lo scopo di identificare le similarità lessicali che intercorrono tra questi.
Il modo più semplice e comune è quello dello shingling ("squamatura"). Un documento non è altro che una stringa di caratteri. Definiamo ***k-shingles*** di un documento ogni sotto stringa di lunghezza k che si trova nel documento. Così facendo, potremmo associare ad ogni documento l'insieme di k-shingles che appaiono in ognuno di essi. Supponiamo che il nostro documento D sia la stringa "*abcdabd*" e fissiamo *k=2*. Allora l'insieme detto *2-shingles* per D sarà {*ab,bc,cd,da,bd*}. Data la natura degli insiemi, la sotto stringa "ab", che

appare due volte nel documento, viene considerata una sola volta.
Il valore di k può essere uno qualsiasi. Tuttavia, se viene scelto un valore di k troppo piccolo, potrebbe capitare che molti k-shingles dei documenti condividano molti elementi. Ad esempio con *k=1*, dato che tutti i documenti utilizzano gli stessi simboli dell'alfabeto, ci troveremo ad avere tutti gli insiemi composti dagli stessi elementi. Quanto grande scegliere k potrebbe dipendere dal grado di similarità che ci siamo prefissati oppure dalla grandezza media dei documenti. L'importante è ricordare che: k dovrebbe essere scelto abbastanza grande in modo tale che la probabilità di ogni shingle di apparire in un documento sia bassa.
Un ulteriore passaggio può essere quello di mappare gli shingles ottenuti in numeri interi (checksum) evitando in questo modo di utilizzare direttamente le sotto stringhe.

$$hash\_checksum : stringa \rightarrow int$$

Firma di documenti L'insieme dei k-shingles ottenuto per ogni documento ha dimensioni molto elevate e perciò è impensabile salvare tale insieme per ogni documento. Dobbiamo, quindi, trovare un modo per replicare questi insiemi in una rappresentazione più piccola chiamata firma (*signature*). La proprietà importante che tale firma deve possedere è quella di poter essere usata per stimare la similarità di Jaccard. Naturalmente avere un esatto valore di similarità è impossibile ma la stima che otterremo è molto vicina al valore reale e più sarà grande la firma più accurata sarà la stima.

Prima di spiegare come sia possibile costruire firme di piccole dimensioni partendo da insiemi di grandi dimensioni, è utile visualizzare una collezione di documenti attraverso la *matrice caratterizzante*. In tale matrice le colonne corrispondono agli insiemi e le righe all'universo degli shingles individuati. Ci sarà un 1 in posizione di riga r e colonna c se l'elemento della riga r è membro dell'insieme della colonna c; altrimenti il valore in posizione *(r,c)* sarà 0.

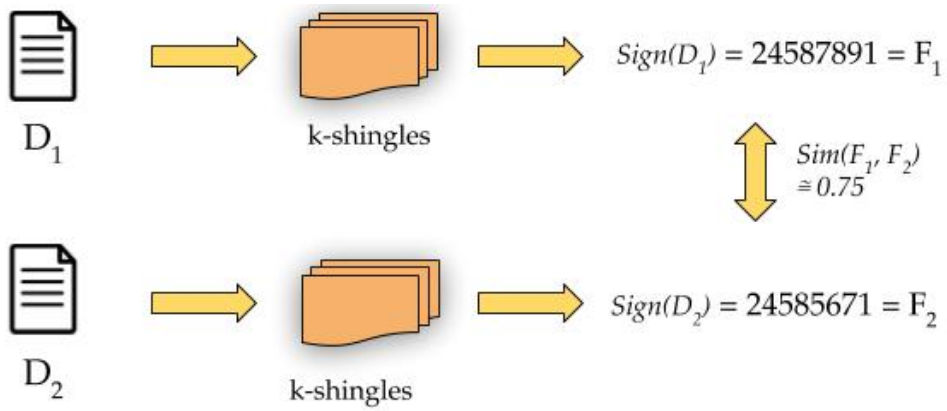


Figura 4.5: Utilizzo della firma

Immaginiamo di avere un insieme formato da 5 shingles {a,b,c,d,e} relativi a 4 documenti fatti in tal modo: $S_1 = \{a, d\}, S_2 = \{c\}, S_3 = \{b, d, e\}$, e $S_4 = \{a, c, d\}$. Nella tabella 4.1 è rappresentata la matrice caratterizzante M relativa ai 4 documenti elencati.

Elemento	S_1	S_2	S_3	S_4
a	1	0	0	1
b	0	0	1	0
c	0	1	0	1
d	1	0	1	1
e	0	0	1	0

Tabella 4.1: Matrice caratterizzante M

È importante ricordare che la matrice caratterizzante non è un modo pe consentire il salvataggio delle informazioni ma ha il solo scopo di visualizzare in maniera più chiara la distribuzione degli shingles.

Minhashing Le firme che vogliamo ottenere per gli insiemi rappresentanti i documenti sono il risultato di una grande quantità di calcoli e ognuno di questi è un "*minhash*" della matrice caratterizzante. Vediamo prima come calcolare questa operazione (minhash) e successivamente come ottenerla in pratica. Per applicare la funzione minhash su un insieme rappresentato da

una colonna della matrice caratterizzante prendiamo una permutazione π delle righe. Il valore minhash di una colonna consiste nel primo numero della riga (nell'ordine di π) in cui la colonna presenta il valore 1. Supponiamo di scegliere una permutazione qualsiasi delle righe $\pi = beadc$ della matrice di tabella 4.1. Tale permutazione definisce una funzione minhash h che mappa un insieme a interi che identificano il numero della riga.

Partendo dalla matrice di tabella 4.1 calcoliamo il valore per l'insieme S_1 secondo la funzione h: la prima colonna ha nelle celle $M(b, S_1)$ e $M(e, S_1)$ il valore 0 mentre la prima cella contenente il valore 1 è $M(a, S_1)$ e perciò $h(S_1) = a$. Continuando per gli altri insiemi avremo i seguenti valori: $h(S_2) = c$, $h(S_3) = b$, $h(S_4) = a$.

Elemento	S_1	S_2	S_3	S_4
b	0	0	1	0
e	0	0	1	0
a	1	0	0	1
d	1	0	1	1
c	0	1	0	1

Tabella 4.2: Matrice M con righe permutate

Generazione della firma Pensiamo nuovamente ad una collezione di insiemi rappresentati dalla matrice caratterizzante M (tabella 4.1). Per rappresentare questi insiemi, prendiamo un numero arbitrario n di permutazioni di righe della matrice M. Chiamiamo la funzione *minhash* h determinata dalle permutazioni h_1, h_2, ..., h_n. Dalla colonna rappresentante un generico insieme S, costruiamo la firma minhash per questo insieme ottenendo il vettore $[h_1(S), h_2(S), ..., h_n(S)]$. Perciò dalla matrice caratterizzante M costruiamo la matrice delle firme SIG in cui l'i-esima colonna di M è replicata con il valore della firma dell'i-esima colonna in SIG. Alla fine otterremo una matrice SIG che avrà le stesse colonne di M ma un numero molto inferiore di righe (dipende dal numero scelto n).
In concreto, non è possibile permutare le righe di una matrice caratterizzante di grandi dimensioni. Persino effettuare una singola permutazione di una

matrice formata da milioni di righe risulta essere computazionalmente troppo costosa. Perciò il metodo descritto prima per la creazione della firma non può essere applicato e/o implementato. Fortunatamente è possibile simulare gli effetti di una permutazione casuale da una funzione hash che mappa numeri di riga in *buckets*. Una funzione hash che mappa interi da 0...k-1 a interi di bucket 0. . . .k-1 tipicamente mapperà qualche coppia di interi in uno stesso bucket lasciando altri buckets vuoti. Tuttavia ciò non è molto importante fintanto che k è grande e non ci sono tante collisioni.
Quindi, anziché scegliere un numero n di permutazioni scegliamo un numero n di funzioni hash $h_1, h_2, ..., h_n$. Costruiamo la matrice delle firme considerando ogni riga nel loro ordine.
A questo punto, sia $SIG(h_i, c)$ l'elemento della matrice delle firme per la funzione hash h_i e la colonna c. Inizialmente fissiamo $SIG(h_i, c)$ a infinito per ogni h_i e c. Consideriamo una riga r della matrice caratterizzante e applichiamo le seguenti operazioni:

1. Calcoliamo $h_i(r)$ per $1 \leq i \leq n$
2. Per ogni colonna c:
   a. se $M(r, c) = 0$ nella riga r, non fare nulla
   b. se $M(r, c) = 1$ allora $\forall h_i$ $SIG(h_i, c)$ è uguale al più piccolo valore tra $SIG(h_i, c)$ e $h_i(r)$

Consideriamo la matrice caratterizzante di tabella 4.1 che viene trasformata nel seguente modo:

Elemento	S_1	S_2	S_3	S_4	$x + 1 \, mod \, 5$	$3x + 1 \, mod \, 5$
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

Tabella 4.3: Matrice caratterizzante M con calcolo funzione hash

Le funzioni hash scelte in questo esempio sono le seguenti:

1. $h_1(x) = x + 1 \, mod \, 5$
2. $h_2(x) = 3x + 1 \, mod \, 5$

Le funzioni prendono in input il numero della riga (checksum dello shingle) e il risultato è inserito nelle ultime due colonne della tabella 4.3. Una vera permutazione attraverso l'utilizzo di una funzione hash si ottiene solo se il modulo è posto con un numero primo (5 nel caso di esempio). Simuliamo ora l'algoritmo per il calcolo della matrice delle firme. Inizialmente la matrice è formata da celle contenenti tutti valori infinito:

	S_1	S_2	S_3	S_4
h_1	∞	∞	∞	∞
h_2	∞	∞	∞	∞

Tabella 4.4: Stato iniziale matrice delle firme *SIG*

Consideriamo la riga 0 della matrice M (tabella 4.1): i valori di $h_1(0)$ e $h_2(0)$ sono entrambi 1. Le celle $M(0, S_1)$ e $M(0, S_4)$ hanno valore 1 e quindi andiamo a modificare solamente le colonne (della matrice delle firme, tabella 4.4) relative agli insiemi S_1 e S_4: siccome 1 è minore di infinito cambiamo entrambi i valori delle colonne S_1 e S_4 ottenendo il seguente stato per la matrice delle firme:

Elemento	S_1	S_2	S_3	S_4
0	1	0	0	1
1	0	0	1	0
2	0	1	0	1
3	1	0	1	1
4	0	0	1	0

Tabella 4.5: Matrice M, riga 0

	S_1	S_2	S_3	S_4
h_1	1	∞	∞	1
h_2	1	∞	∞	1

Tabella 4.6: Stato matrice *SIG*

Ora consideriamo la riga numero 1 della matrice M: solo l'insieme S_3 contiene il valore 1 con $h_1(1) = 2$ e $h_2(1) = 4$. Allora modifichiamo la cella $SIG(h_1, S_3)$ con 2 e la cella $SIG(h_2, S_3)$ con 4. Tutte le altre celle rimangono invariate in quanto nella riga 1 solo S_3 possiede la cella con valore pari a 1.

Elemento	S_1	S_2	S_3	S_4
0	1	0	0	1
1	0	0	1	0
2	0	1	0	1
3	1	0	1	1
4	0	0	1	0

Tabella 4.7: Matrice M, riga 1

	S_1	S_2	S_3	S_4
h_1	1	∞	2	1
h_2	1	∞	4	1

Tabella 4.8: Stato matrice *SIG*

La riga numero 2 ha valore 1 nelle celle $M(2, S_2)$ e $M(2, S_4)$ con $h_1(2) = 3$ e $h_2(2) = 2$. Siccome i valori della matrice SIG per S_4 sono inferiori dei valori delle funzioni hash $h_1(2)$ e $h_2(2)$ la colonna S_4 di SIG non viene alterata. Invece la colonna S_2 viene aggiornata con i valori $h_1(2)$ e $h_2(2)$ in quanto entrambi minori dell'attuale contenuto delle celle della colonna S_2.

Elemento	S_1	S_2	S_3	S_4
0	1	0	0	1
1	0	0	1	0
2	0	1	0	1
3	1	0	1	1
4	0	0	1	0

Tabella 4.9: Matrice M, riga 2

	S_1	S_2	S_3	S_4
h_1	1	3	2	1
h_2	1	2	4	1

Tabella 4.10: Stato matrice *SIG*

Per la riga numero 3 solo S_2 ha valore pari a 0 perciò non subirà alcuna eventuale modifica nella sua relativa colonna della matrice SIG. I valori per la riga 3 delle funzioni hash sono $h_1(3) = 4$ e $h_2(3) = 0$. Il valore 4 non va bene per nessuna delle colonne interessate mentre il valore 0 è minore del contenuto di $SIG(h_2, S_1)$, $SIG(h_2, S_3)$ e $SIG(h_2, S_4)$,

Elemento	S_1	S_2	S_3	S_4
0	1	0	0	1
1	0	0	1	0
2	0	1	0	1
3	1	0	1	1
4	0	0	1	0

Tabella 4.11: Matrice M, riga 3

	S_1	S_2	S_3	S_4
h_1	1	3	2	1
h_2	0	2	0	0

Tabella 4.12: Stato matrice *SIG*

Infine consideriamo l'ultima riga. I relativi valori delle funzioni hash applicati a questa riga sono $h_1(4) = 0$ e $h_2(4) = 3$. Dal momento che solo la cella $M(4, S_3) = 1$ dobbiamo interessarci solamente alla colonna S_3 del matrice delle firme. La colonna della matrice SIG per S_3 contiene i valori $[2, 0]$ che dobbiamo confrontare con quelli delle funzioni hash $[0, 3]$ e quindi la sola cella che viene modificata è quella $SIG(h_1, S_3)$ in quanto $0 < 2$.

Elemento	S_1	S_2	S_3	S_4
0	1	0	0	1
1	0	0	1	0
2	0	1	0	1
3	1	0	1	1
4	0	0	1	0

Tabella 4.13: Matrice M, riga 4

La matrice finale delle firme finale ottenuta è la seguente:

	S_1	S_2	S_3	S_4
h_1	1	3	0	1
h_2	0	2	0	0

Tabella 4.14: Stato finale matrice *SIG*

Le firme ottenute sono le seguenti:

- $Minhash(S_1) = [1\,0]$
- $Minhash(S_2) = [3\,2]$

- $Minhash(S_3) = [0\,0]$
- $Minhash(S_4) = [1\,0]$

Questo esempio ha semplicemente lo scopo di mostrare il funzionamento dell'algoritmo della generazione delle firme. In un caso reale avremmo un numero n di funzioni hash molto maggiore in modo da avere una stima il più possibile vicina al valore reale. Infatti, come vedremo nela capitolo 5 e nello specifo nella sezione relativa l'implementazione di Minhash, verranno definiti due vettori di dimensione N (grandezza della firma) che verranno utilizzati per identificare le N funzioni hash scelte.

Minhashing e la similarità di Jaccard C'è una connessione tra il risultato dell'algoritmo appena mostrato e la similarità di Jaccard: la probabilità che la funzione minhash produca lo stesso valore per due insiemi è uguale alla similarità di jaccard tra questi due insiemi. Per vedere come sia possibile, abbiamo bisogno della matrice caratterizzante M di tabella 4.1, considerando gli insiemi S_1 e S_2. Le righe possono essere divise in tre classi:

1. Tipo X, quelle righe che hanno 1 in entrambe le colonne;
2. Tipo Y, quelle righe che hanno 1 in una colonna e 0 nell'altra;
3. Tipo Z, quelle righe che hanno 0 in entrambe le colonne.

Dal momento che la matrice è sparsa, la maggior parte delle righe saranno di tipo Z. Tuttavia, il rapporto tra il numero di righe di tipo X e Y è determinato sia dalla formula per la similarità di Jaccard $SIM(S_1,S_2)$ e sia dalla probabilità che $h(S_1) = h(S_2)$. Sia x il numero di righe di tipo X e y il numero di righe di tipo Y. Allora $SIM(S_1,S_2) = x/(x+y)$ in quanto $x = |S_1 \cap S_2|$ *e* $x + y = |S_1 \cup S_2|$. Consideriamo ora la probabilità che $h(S_1) = h(S_2)$. Con le righe permutate casualmente e procedendo dall'alto, la probabilità di incontrare una riga di tipo X prima di una di tipo Y è pari a $x/(x+y)$. Ma se la prima riga è di tipo X allora sicuramente $h(S_1) = h(S_2)$. Mentre se la prima riga è di tipo Y allora l'insieme contenente il valore 1 nella propria colonna assumera' il relativo valore della funzione minhash. Tuttavia

se l'insieme nella colonna ha valore 0 nella riga corrente, assumerà qualche altro valore in base alla permutazione. Quindi se la riga è di tipo Y avremo che $h(S_1) \neq h(S_2)$ mentre se la riga è di tipo X avremo $h(S_1) = h(S_2)$ con probabilità $x/(x+y)$ che corrisponde alla similarità di Jaccard.

Torniamo all'esempio e calcoliamo la similarità di Jaccard tra le firme mettendole in paragone con i valori reali. Le firme di S_1 e S_4 sono identiche e quindi $SIM(Minhash(S_1), Minhash(S_4)) = 1.0$ mentre il vero rapporto di similarità tra S_1 e S_4 considerando la matrice caratterizzante è $2/3$. La $SIM(Minhash(S_1), Minhash(S_3)) = 1/2$ mentre quella effettiva è di $1/4$; $SIM(Minhash(S_1), Minhash(S_2)) = 0$ che coincide con quella reale.

Perciò possiamo affermare che dati due oggetti x e y appartenenti ad una collezione, *Minhash* è una buona funzione hash che permette di applicare lo schema LSH:

$$P[minhash(x) = minhash(y)] = sim(x, y)$$

dove $sim(\cdot, \cdot)$ rappresenta la *Jaccard similarity*.

Tecnica del banding Anche se ora possiamo usare *minhashing* per comprimere i documenti in firme di piccole dimensione preservando la similarità prevista per ogni coppia di documenti, potrebbe comunque non bastare per effettuare un'efficiente ricerca. Supponiamo di avere un milione di documenti ognuno dei quali rappresentato da una firma di lunghezza pari a 250. Il numero di coppie di documenti da confrontare è pari a $\binom{1\,000\,000}{2}$. Se ci volesse un microsecondo per calcolare la similarità tra due firme, allora servirebbero quasi sei giorni per calcolare tutte le coppie.
Se il nostro obiettivo è quello di calcolare le similarità tra tutte le possibili coppie di documenti, non c'è nulla che possa ridurre il lavoro, se non con il parallelismo. Tuttavia, in questo lavoro di tesi, siamo interessati a trovare tutti quei documenti che soddisfano un certo grado di similarità con un nuovo punto in input. LSH permette di concentrarci su questo aspetto. Una volta ottenuta la firma per ogni documento, attraverso l'applicazione di minhash, un modo efficace di procedere è quello di dividere la matrice delle firme *SIG* (tabella 4.14) in b *bands* ognuno composto da r righe. Per

ogni band, scegliamo una funzione hash che mappa vettori di r interi in un numero che ne identifica l'indirizzo nella tabella hash. Possiamo utilizzare la stessa funzione hash per tutti i bands ma è conveniente applicare un range diverso (ad ogni band) in modo che differenti righe di band, anche avendo stessi valori, non finiscano in bucket uguali. Così facendo, possiamo analizzare ogni coppia memorizzata nello stesso bucket considerando questi elementi *coppie candidate*. La speranza è che la maggior parte delle coppie differenti non venga memorizzata nello stesso bucket. Se ciò accade, chiamiamo queste coppie *falsi positivi*. D'altra parte, vorremmo che le coppie simili vengano memorizzate nello stesso bucket da almeno una funzione hash. Se ciò non accade, chiamiamo queste coppie *falsi negativi*.

La tabella 4.15 mostra una parte della matrice delle firme di 12 righe raggruppate in 4 bands da 3 righe ciascuno. La seconda e la quarta colonna hanno gli stessi valori [0 2 1] nel band 1, e quindi verranno memorizzati nello stesso bucket. Pertanto, indipendentemente dal valore delle altre colonne negli altri 3 bands, questa coppia sarà considerata una coppia di candidati. È possibile che anche altre colonne del band 1 vengano memorizzate nello stesso bucket. Tuttavia, dal momento che le colonne hanno valori differenti, [0 2 1] e [1 3 0], e ci sono altri bucket disponibili, la probabilità di una collisione è molto bassa. Per semplicità assumiamo che due vettori sono memorizzati nello stesso bucket se e solo sono identici. Due colonne differenti nel band 1 hanno altre 3 occasioni per diventare una coppia di candidati potrebbero essere identici in uno degli altri bands. Comunque possiamo osservare che più due colonne sono simili, più alta sarà la probabilità di essere memorizzati nello stesso bucket.

band 1	...	1 3 0	0 2 1	0 1 3	0 2 1	2 2 1	...
band 2							
band 3							
band 4							

Tabella 4.15: Divisione della matrice delle firme in bands

Analisi della tecnica del banding Supponiamo di usare b bands e di r righe ciascuno e di avere una coppia di documenti con valore di similarità di Jaccard pari a s. Calcoliamo la probabilità che questi due documenti (attraverso l'utilizzo delle loro firme) diventino una coppia di candidati, nel seguente modo:

1. La probabilità che le firme siano identiche in tutte le righe di un band è s^r
2. La probabilità per due band di essere differenti è $1 - s^r$
3. La probabilità che le firme non abbiano nemmeno band in comune è $(1 - s^r)^b$
4. La probabilità che le firme abbiano almeno un band in comune e quindi tali firme siano una coppia di candidati è $1 - (1 - s^r)^b$

A prescindere dai valori scelti, tale funzione in b e r ha la forma di una *S-curve*. Il valore soglia nella figura 4.6, *threshold*, rappresenta il grado di similarità s in cui la probabilità di diventare un candidato è $1/2$.

Consideriamo il caso di $b = 20$ e $r = 5$ con lunghezza della firma pari a 100. Dividiamo in 20 bands, 5 righe ciascuno, la matrice delle firme. Nella

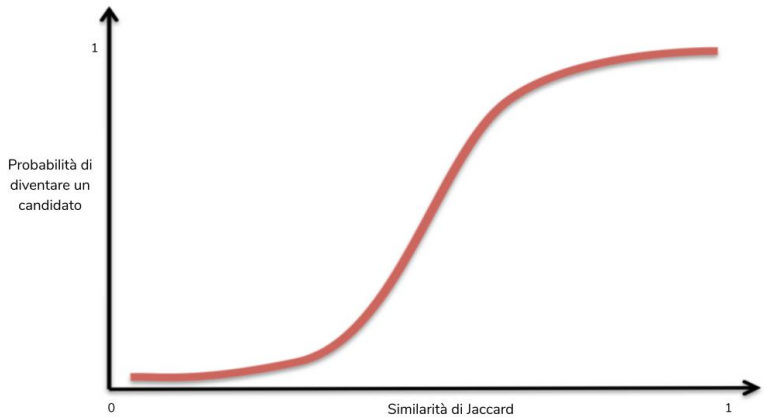


Figura 4.6: Esempio di un problema NN.

tabella 4.16 sono riportati alcuni valori della funzione $1-(1-s^5)^{20}$. Si nota che il valore di soglia, valore di s in cui la curva è in risalita, è leggermente maggiore di 0.5 e che la curva cresce più velocemente nell'intervallo $[0.4, 0.6]$.

s	$1-(1-s^r)^b$
0.2	0.006
0.3	0.047
0.4	0.186
0.5	0.470
0.6	0.802
0.7	0.975
0.8	0.999

Tabella 4.16: Valori della curva per $b = 20$ e $r = 5$

4.3.2 Funzioni LSH

Lo scopo di questa parte delle dissertazione è quello di chiarire il significato delle *locality sensitive functions*. Le tecniche LSH viste fino ad ora sono due esempi di famiglie di funzioni che permettono di individuare coppie di candidati in base al valore della distanza tra questi. La figura 4.6 riflette come poter effettivamente evitare il problema dei falsi positivi e negativi.

Vediamo ora, in linea teorica, quali sono le tre condizioni necessarie affinché una famiglia di funzioni sia considerata applicabile alla tecnica LSH:

1. la probabilità di determinare due elementi come candidati deve essere maggiore per elementi che hanno distanza minore;

2. le funzioni devono essere statisticamente indipendenti nel senso che deve essere possibile stimare la probabilità che due o più funzioni diano un certo valore partendo dal prodotto di eventi indipendenti;

3. le funzioni devono essere efficienti in due modi:

   - devono essere in grado di determinare i candidati in un tempo molto inferiore rispetto al tempo di verifica di tutte le coppie. Per esempio, le funzioni minhash hanno questa capacità, dal momento che possiamo mappare insiemi in valore minhash in tempo proporzionale alla dimensione dei dati. Dato che gli insiemi con valori comuni sono collocati nello stesso bucket, considereremo solo quegli elementi che condividono un bucket (complessità sublineare);
   - devono permettere di essere combinate per costruire funzioni che evitino falsi negati e positivi. Ad esempio, attraverso la tecnica del banding siamo in grado di diminuire il numero di elementi errati andando a concatenare diverse funzioni ottenendo una funzione con andamento simili a quello mostrato in figura 4.6.

Consideriamo funzioni che prendono due oggetti in input e restituiscono una scelta sul fatto che gli oggetti possano essere candidati. Useremo la notazione $f(x) = f(y)$ per indicare che "x e y sono una coppia di candidati', mentre useremo $f(x) \neq f(y)$ per indicare che "x e y non sono una coppia di candidati per questa funzione f". Siano d_1 e d_2 distanze secondo una qualche metrica. Una famiglia $\mathbb{F}$ di funzioni è $(d_1, d_2, p_1, p_2) - sensitive$ se per ogni $f \in \mathbb{F}$:

1. se $d(x, y) \leq d_1$, allora la probabilità che $f(x) = f(y)$ è almeno p_1;

2. se $d(x, y) \geq d_2$, allora la probabilità che $f(x) = f(y)$ è al massimo p_2.

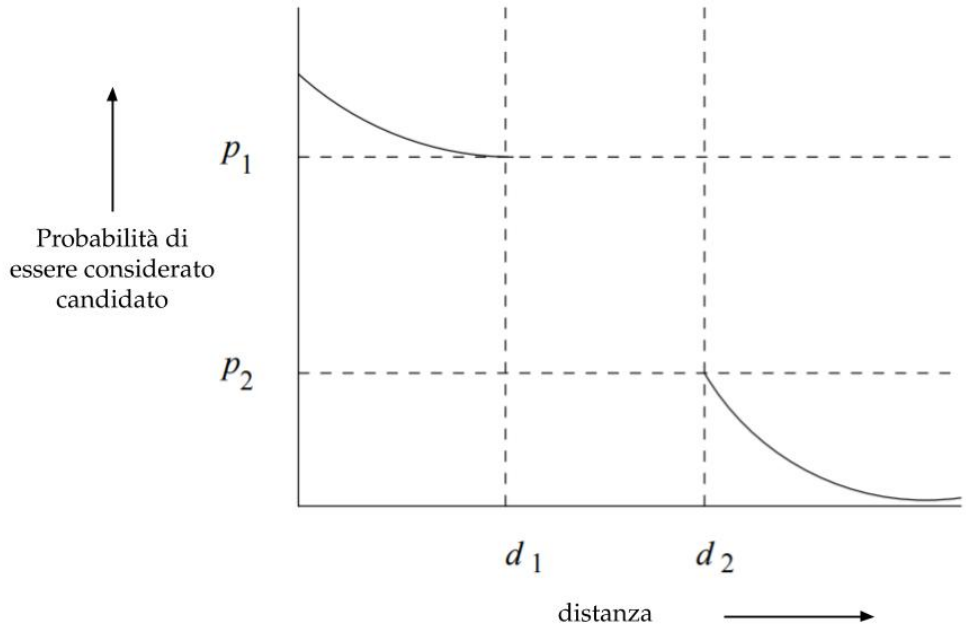


Figura 4.7: Comportamento di una funzione $(d_1, d_2, p_1, p_2) - sensitive$

La figura 4.7 mostra l'andamento della funzione $(d_1, d_2, p_1, p_2) - sensitive$ in base ad una certa distanza e probabilità. Dalla figura si può notare che il comportamento della funzione per valori di $x \in [d_1, d_2]$ non è descritto ma potremmo assegnare a d_1 e d_2 valori molto simili ma otterremmo, di conseguenza, dei valori molto simili anche per p_1 e p_2.

Per fare chiarezza consideriamo la famiglia delle funzioni minhash con la relativa distanza di Jaccard: una funzione h considera x e y una coppia di candidati se e solo se $h(x) = h(y)$.

Le minhash functions sono una famiglia di funzioni $(d_1, d_2, p_1, p_2)-sensitive$ *per ogni valore di* d_1 *e* d_2*, dove* $0 \leq d_1 \leq d_2 \leq 1$.

Se $d(x, y) \leq d_1$, dove d è la distanza di Jaccard, allora $SIM(x, y) = 1 - d(x, y) \geq 1 - d_1$. Come sappiamo, la *Jaccard similarity* tra x e y è uguale alla probabilità per la funzione minhash per x e y di produrre lo stesso valore.

Consideriamo $d_1 = 0.3$ e $d_2 = 0.6$. Possiamo assumere che le famiglia di funzioni minhash è una famiglia di funzioni $(0.3, 0.6, 0.7, 0.4) - sensitive$. Se la distanza di Jaccard tra x e y è al massimo 0.3 allora la probabilità che x e

y abbiano uno stesso valore, derivante da una funzione minhash, è pari a 0.7. Mentre se la distanza di Jaccard tra x e y è minimo 0.6 la probabilità che x e y abbiamo lo stesso valore è 0.4. L'unico requisito richiesto è che $d_1 < d_2$.

4.4 Multi-Reference Cosine Text Algorithm

Nella sezione 3.2 è stato presentato un modo che permette di eseguire la *vectorization* di un testo. Uno dei maggiore svantaggi della tecnica *TF-IDF* è la dimensionalità del vettore generato: lineare al numero di elementi individuati nel corpus di partenza. Esistono delle tecniche che permettono di ridurre la dimensionalità della firma e LSH è una di quelle. In questa sezione verrà presentato un algoritmo la cui finalità è quella di risolvere il problema della dimensionalità derivante da *TF-IDF*. Negli articoli [MN18] e [MK19] gli autori presentano un nuovo algoritmo per la *detection* dei duplicati. Per generare la firma di ogni documento, la tecnica proposta confronta il testo in input D_i con ogni parte del testo di riferimento, o *reftext*. Ogni confronto produce un valore decimale $x \in [0, 1]$ derivante dal calcolo della *cosine similarity*. L'algoritmo, successivamente, prende questi valori decimali per creare il vettore che verrà utilizzato come firma del documento D_i. Come detto nel caso della firma generata da LSH, anche qui il vettore risultate ha la proprietà di preservare la similarità iniziale dei documenti di partenza. Pertanto *MRCTA* può essere utilizzato come sistema per rilevare gli DND. La figura 4.8 mostra la struttura complessiva con l'utilizzo dell'algoritmo descritto. Il *reftext* è una delle componenti principali del multi-reference cosine text algorithm. Il testo di riferimento è una sequenza di 3-words che viene utilizzato dall'algoritmo per la generazione della firma di ogni documento. Se da un lato considerare n-grammi di dimensione maggiore può essere piu efficace in termini di misurazione della similarità o differenza tra due testi, dall'altro un valore molto grande di n implica un numero maggiore di confronti. Detto questo, nell'articolo [MK19] e dai relativi test effettuati è emerso che per $n=3$ si ottengono buoni risultati in termini di similarità e performance.

Per generare la firma, *MRCTA* divide il *reftext* in diverse partizioni. L'algoritmo, poi, confronta un documento con le diverse parti del *reftext* usando

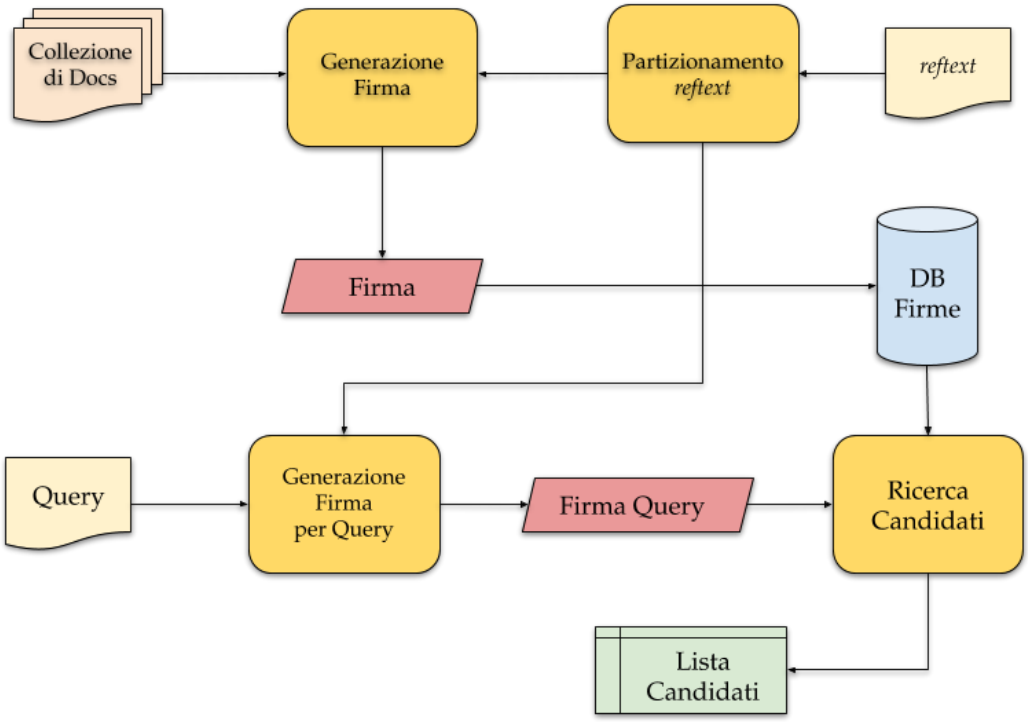


Figura 4.8: Stuttura del sistema con l'utilizzo di MRCTA

la *cosine similarity*. In altre parole, viene valutata la frequenza dei 3-words della partizione considerata del reftext con l'intero documento in input. Per tale motivo, i migliori 3-words devono essere considerati come componenti per la creazione del *reftext*. Pertanto, l'assenza o la presenza di un 3-word nel reftext potrebbe avere un considerevole impatto sulla generazione finale della firma. Detto questo, consideriamo il seguente esempio: sia dato un *reftext* contenente un certo insieme di 3-words. Se l'algoritmo dovesse confrontare un documento con questo un unico reftext il risultato finale potrebbe generare delle firme che non rispecchia la vera similarità tra documenti. Immaginiamo che il reftext considerato sia formato dai due insiemi di trigammi $3-grams_n$ e $3-words_m$. Sia dato un documento D_i che contiene $3-words_n$ e che quindi avrà un alto valore di similarità con una parte e zero similarità con l'altra. Sia preso in considerazione un ulteriore documento D_j che contiene i $3-words_m$ e che quindi avrà un alto valore di similarità con la parte contenente $3-words_m$ e zero similarità con l'altra. Questi due documenti avranno

la medesima firma in quanto il loro contenuto è formato dello stesso numero di 3-words ma relativi ad insiemi diversi e quindi saranno considerati simili anche se $SIM(D_i, D_j) = 0.0$.

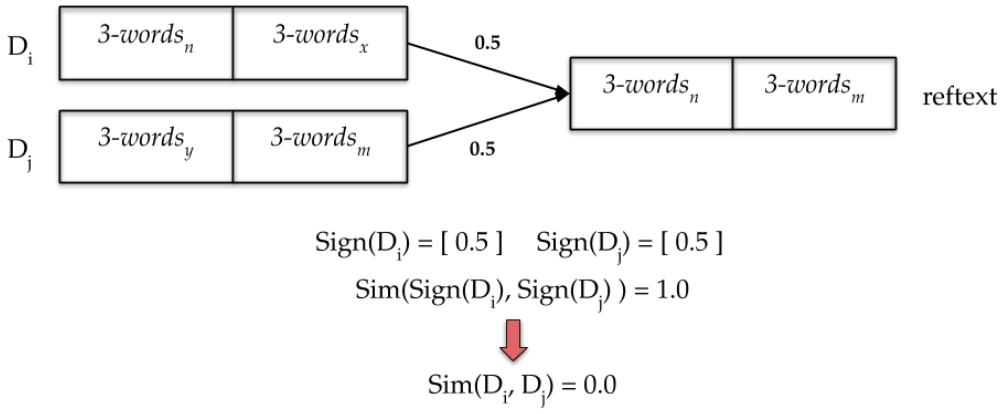


Figura 4.9: Calcolo firma con un solo reftext

Dividendo il reftext in più parti, ognuna contenente un numero fissato di 3-words, l'algoritmo potrebbe ottenere più informazioni da ognuna di queste e quindi una maggiore accuratezza. Se il reftext fosse formato da tutti i possibili 3-words del corpus di partenza e riducendo la dimensione di ogni partizione al singolo 3-word, l'algoritmo creerebbe un vettore identico a quello che otterremo con la tecnica TF-IDF. In tal modo possiamo affermare che minore è la dimensione di ogni partizione e maggiore sarà l'accuratezza finale. Tuttavia, con quest'ultimo modo di procedere le performance si avvicinano molto ad un approccio brute force. Non tutti i 3-words hanno la stessa importanza e potremmo avere dei 3-words con più informazione rispetto ad altri. Considerando solo i 3-words più importanti per la costruzione del reftext, l'algoritmo *MRCTA* può ridurre drasticamente la dimensione del testo di riferimento con una conseguente miglioria delle performance. Il numero esatto di partizioni dipende dall'accuratezza desiderata e dal tempo di calcolo che si è disposti a concedere. Il reftext ha un significativo impatto sulle performance dell'algoritmo appena descritto e la generazione di un buon reftext è una delle parti più critiche per il raggiungimento di buoni risultati. Nell'articolo [MK19] è utilizzato un algoritmo genetico per la formazione del

reftext ma nel corrente lavoro di tesi ci fermeremo solo alla fase di *popolazione iniziale* che prende in considerazione gli N 3-words con score TF-IDF più alto. Bisogna tenere in considerazione che più è grande la dimensione del reftext e migliore sarà l'accuratezza ma questo produrrà una drastica caduta delle performance.

In questa sezione è stato affrontato il problema della ricerca degli duplicati e quasi duplicati (DND). Abbiamo visto come le tecniche proposte per la risoluzione del problema citato siano utili all'obiettivo di questa tesi: progettare un sistema efficiente per il recupero di elementi simili in base ad una determinata metrica. Principalmente le tecniche viste sono state due: *Locality Sensitive Hashing* e *Multi-Reference Cosine Text Algorithm*. L'idea alla base di LSH è quella di definire una famiglia di funzioni che applicata a punti vicini inserisca nello stesso bucket questi due punti e se tali punti sono lontani inserisca tali punti in bucket diversi. Per evitare bucket vuoti e quindi per diminuire la probabilità di restituire una soluzione vuota, possiamo creare più hash tables in modo da mappare i punti in più bucket. Questa soluzione genera troppi candidati e quindi è stato introdotto il problema dell'Approximate Nearest Neighbor Search (ANN). Quindi le funzioni LSH devono garantire la proprietà che:

- se $u \in B(q, r)$ allora $Pr[h(u) = h(q)] \geq \alpha$;
- se $u \notin B(q, R)$ allora $Pr[h(u) = h(q)] \leq \beta$

con h casualmente scelto dalla famiglia di funzioni considerata e $R = r(1+\epsilon)$ detto fattore di approssimazione. In altre parole se u è nello stesso bucket della query q significa che tra di loro c'è una relazione di similarità di almeno r e quindi la probabilità che entrambi finiscano nello stesso bucket è almeno α. È stata presa in esame uno schema di LSH chiamato *Minhash* che permette di stimare la distanza Jaccard attraverso l'utilizzo delle firme generate. In altre parole, il valore di similarità ottenuto dal confronto di due firme relative a due documenti È una stima del valore che otterremo andando a confrontare i due documenti originali. Ad esempio, con Minhash avremmo che: $Jacc(D_1, D_2) \approx Jacc(Minhash(D_1), Minhash(D_2))$.

L'ultima tecnica descritta (*MRCTA*) è, in poche parole, un'ottimizzazione di TF-IDF. *MRCTA*, infatti, permette di definire a priori la lunghezza della firma desiderata rendendo tale lunghezza indipendente dal numero di parole contenute nel dizionario e riducendo in maniera drastica il numero di dimensioni del vettore rappresentate il documento in input. Naturalmente, la lunghezza della firma e il modo in cui è generata influiscono pesantemente sia sull'accuratezza che sulle performance del sistema: avere una firma più grande migliora l'accuratezza ma ne diminuisce le performance; avere una firma piccola migliora le performance ma diminuisce l'accuratezza.

Nel capitolo successivo verranno descritti i vari aspetti riguardanti la fase di normalizzazione ed estrazione delle varie parti considerate, le tecniche implementate e un ambiente di sviluppo e test in cui poter confrontare i vari approcci utilizzati.

Capitolo 5

Similarity Hashing Enviroment

Nei capitoli precedenti sono stati affrontati concetti fondamenti per la comprensione del capitolo corrente nel quale verrà spiegato il contributo offerto come progetto di tesi. Fino ad ora sono state descritte, in linea teorica, tecniche che consentono di quantificare la similarità lessicale (capitolo 2) di due testi e, al fine di evitare una ricerca lineare, spesso infattibile, necessitano tecniche che riducano la dimensionalità dei documenti. È stato visto un possibile approccio che utilizza lo schema chiamato LSH (sezione 4.3) in modo da poter rappresentare e descrivere un documento attraverso la generazione di una firma ottenuta iterativamente elaborando gli *shingles* del documento stesso. La proprietà fondamentale che questa firma deve possedere è quella relativa al mantenimento della similarità del documento con un altro. Sulla base di questo concetto è stato analizzato e dimostrato come l'utilizzo dell'algoritmo *Minhash* (sezione 4.3.1) possa permettere il calcolo di una stima della distanza di *Jaccard*. Generata la firma, abbiamo visto come, tramite il suo utilizzo, si possa risolvere il problema della ricerca lineare con la cosiddetta tecnica del *Banding*. Oltre lo schema LSH è stato analizzato un ulteriore algoritmo che permette ridurre il numero di dimensioni del vettore che normalmente è utilizzato con un approccio TF-IDF (sezione 3.2). Con *multi-reference cosine text* (sezione 4.4) abbiamo visto come si possa decidere, a priori, la dimensione del vettore rappresentante il documento.

Tutto ciò è stato visto a livello teorico e quindi in questo capitolo vedremo come poter utilizzare, testare e confrontare algoritmi e approcci diversi

partendo da un dataset di riferimento. Partendo da questa collezione di documenti applicheremo delle operazioni di normalizzazione ed estrazione creando fino a 4 ulteriori dataset (uno per ogni parte interessata) che saranno gli input per la creazione di ogni tecnica presa in considerazione. Ogni tecnica potrà essere interrogata attraverso l'implementazione di un'API il cui indirizzo verrà messo a disposizione in modo da renderla accessibile e confrontabile con altre tecniche in gioco. Avendo a disposizione tutti gli *entrypoint* delle tecniche è stato creato un ambiente (*SHE*) che consente l'interrogazione simultanea di ogni API. In figura 5.1 sono mostrate le fasi per la creazione e l'inserimento di una tecnica X all'interno dell'ambiente SHE.

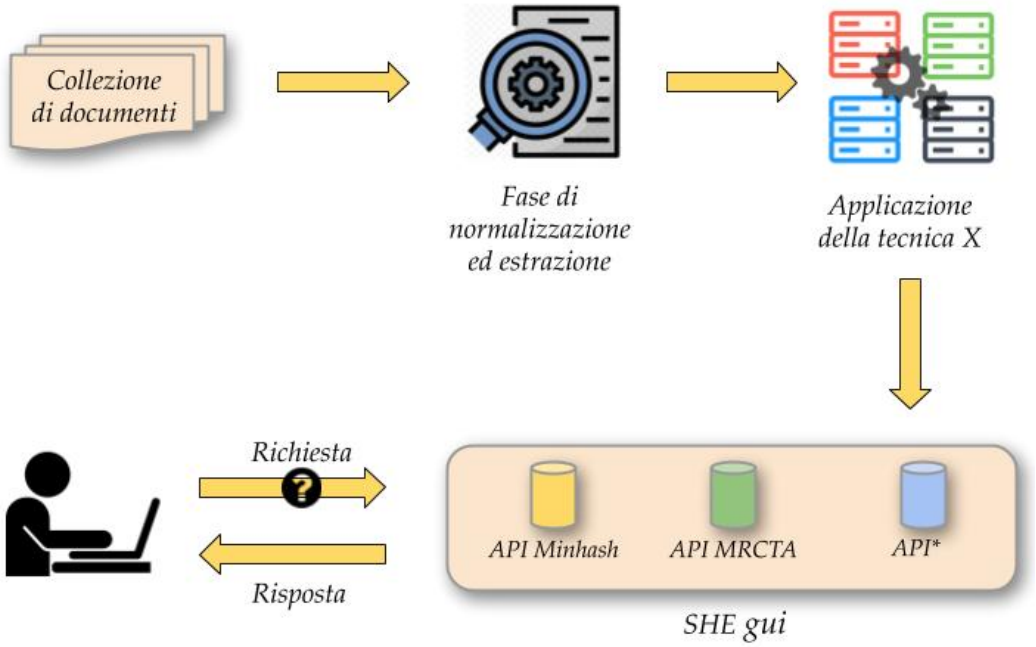


Figura 5.1: Fasi per la creazione e l'utilizzo di una tecnica

Il capitolo inizia con la descrizione del dataset e le relative operazioni di normalizzazione ed estrazione. Successivamente vengono definiti i modelli implementati e di come sia possibile rendere accessibile una nuova tecnica: verrà definito uno standard che ogni tecnica ha seguito per integrarsi all'ambiente SHE il quale verrà descritto nell'ultima parte di questo capitolo ed utilizzato nella fase di testing (capitolo 6).

5.1 Dataset

Come ambito per la costruzione di una ipotetica collezione di documenti di partenza è stato scelto quello legislativo. I documenti legislativi seguono uno schema comune: vi è una parte iniziale (preambolo) che descrive l'introduzione agli articoli posti nel resto del documento. Ogni sezione descrive un articolo e sono presenti paragrafi e frasi che condividono dei pattern comuni. Si è quindi deciso di utilizzare documenti provenienti dal sito *Eur-lex* (sezione *Ambiente, consumatori e protezione della salute*[1]), tutti in lingua in inglese e in formato HTML che formano una collezione di 1800 documenti. Di questi, una parte composta da 1500 documenti è stata utilizzata per la fase di sviluppo mentre i restanti 300 sono stati utilizzati per la costruzione di query da utilizzare per la fase di testing.

5.1.1 Fase di estrazione

Ottenuta la collezione di documenti, si è passati alla fase di estrazione. In questa fase sono stati individuati per ogni documento della collezione di sviluppo frammenti di testo relativi a:

- Sezione;
- Paragrafo;
- Frase;
- 3-words.

Essendo ogni documento in formato HTML, l'operazione di estrazione del testo inerente a ogni frammento di tipo sezione è stata abbastanza facile e infatti sono stati considerati tutte quelle parti di documenti identificate dal tag $< section >< /section >$. Per individuare i frammenti paragrafi è stata adottata la seguente regola: preso il testo del documento (compreso di tag HTML) ogni paragrafo è identificato dalla combinazione $. < /p >$. Questo perchè nei documenti, il testo tra i tag $< p >$ non sempre identifica

[1] https://eur-lex.europa.eu/browse/directories/legislation.html

la chiusura grammaticale del paragrafo ma l'utilizzo del tag in questione è spesso utilizzato come meccanismo per l'andata a capo. Per le frasi, sono considerate tali ogni sequenza di testo delimitata da punti, punti e virgola e due punti. I 3-words, non sempre applicabili (dipende dall'algoritmo utilizzato), sono stati ricavati partendo dalle frasi. È considerato un 3-word ogni sequenza di tre parole rilevanti. Questo per evitare di considerare come 3-words sequenze di testo formate da soli articoli, preposizioni, avverbi. Ad esempio, se consideriamo la frase *"Il treno per Bologna partirà in orario"* otteniamo i due seguenti 3-words:

1. { treno, Bologna, partirà }
2. { Bologna, partirà, orario }

È a discapito del programmatore (ossia chi implementa la tecnica) definire delle regole che portino a giudicare cosa è o non è rilevante. Nella frase precedente sono state rimosse le cosiddette *stopwords* (il, per, in). Sulla base di ciò che è stato descritto in questa sottosezione, possiamo riassumere gli elementi relativi al dataset considerato attraverso la tabella 5.1.1.

	Env_{Set}	$Test_{Set}$
Documenti	1535	300
Sezioni	9096	680
Paragrafi	30811	2375
Frasi	63001	4357
3-words	10^6	-

I 3-words sono estrapolati direttamente dalle frasi, questo per permettere una manipolazione di questi più accurata. Ogni elemento è stato poi etichetto in modo da poter risalire al documento di appartenenza.

5.1.2 Fase di normalizzazione

La fase di normalizzazione, o pulitura, viene eseguita sia sugli elementi dell'envset e sia sulle query provenienti dall'ambiente SHE. L'obiettivo di questa fase è quello di rimuove parti irrilevanti del testo o renderle nella loro forma base. Ad esempio, vorremmo che il sistema ci riconosca come identiche le parole *parlare* e *parla* e per fare ciò abbiamo bisogno di definire un elenco di operazioni (una *pipeline*) che possano permettere questa e altre casistiche. Le operazione di normalizzazione a cui un testo in input si sottopone sono le seguenti:

1. rimozione di spazi superflui; senza di questo un algoritmo potrebbe identificare come diverse le strighe "hello " da "hello";
2. sostituzione con dei *placeholder* di pattern speciali come:
    - date, sia nel formato esteso "20 Settembre 2011" che nella forma abbreviata "20/09/11" con placehoder $< date >$;
    - numeri tra parentesi ("(10)") con placeholder $< numpar >$;
    - elenchi puntati e numeri ("1.") con placeholder $< num >$;
    - numeri che identificano una legge (20/2010), con placeholer $< numlex >$;
    - nomi di stati (*Italy*, *United States*,...) con placeholder $< country >$;
3. espansione delle seguentu 10 abbreviazioni più comuni;
    - CEN: 'European Committee for Standardisation',
    - EEC: 'European Economic Commission',
    - 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'

4. rimozione punteggiatura e stopword;
5. *lemming* ossia il processo per ridurre una parola nella sua forma canonina (il verbo "parliamo" diventa "parlare");
6. conversione in lowercase;
7. divisione in 3-word.

Il riepilogo e l'ordine delle operazioni appena elencate è mostrato nella figura in basso.

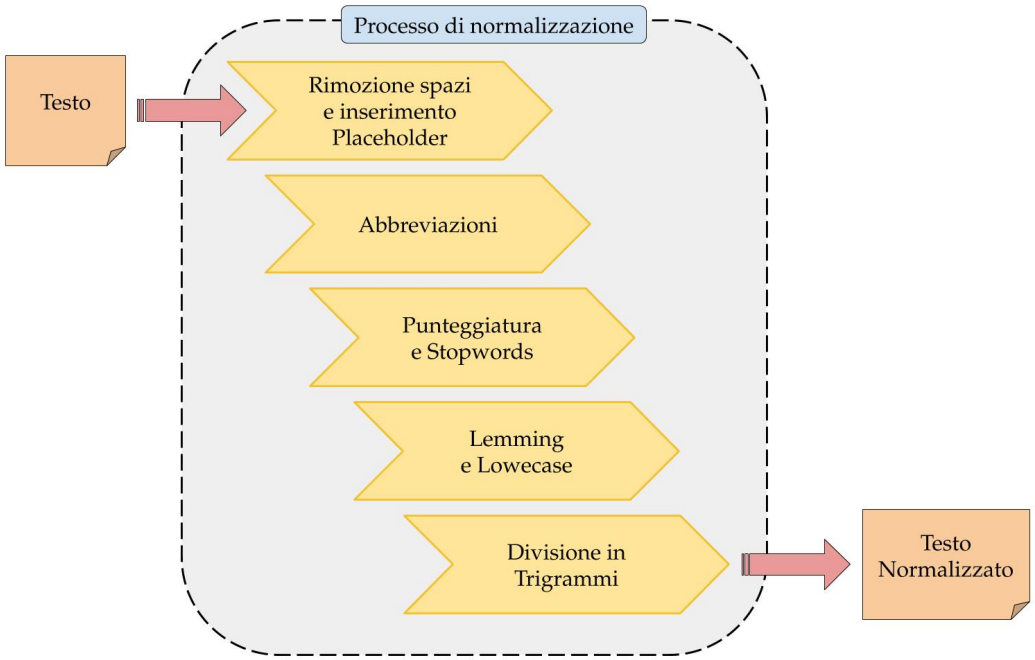


Figura 5.2: Processo di normalizzazione

Lo scopo della fase di *lemmatization* è di ridurre le forme flesse a una forma base comune. Al contrario dello *stemming*, la lemmatizzazione non taglia semplicemente le inflessioni ma utilizza le basi di conoscenza lessicale per ottenere le forme di parole di base corrette. Con lo scopo di avere

ben chiari gli output di queste operazioni, prendiamo in considerazione la seguente frase e il suo relativo processo di normalizzazione:

In Italy, the specific chemical requirements laid down by this directive should aim at protecting the health of children from certain substances in toys, while the 18 environmental concerns presented by toys are addressed by horizontal environmental legislation applying to electrical and electronic toys, namely directive 2002/95/EC of the European parliament and of the council of 27 january 2003.

rimozione spazi superflui e sostituzione pattern

*In <**COUNTRY**>, the specific chemical requirements laid down by this directive should aim at protecting the health of children from certain substances in toys, while the <**NUM**> environmental concerns presented by toys are addressed by horizontal environmental legislation applying to electrical and electronic toys, namely directive <**NUMLEX**>/EC of the European parliament and of the council of <**DATE**>.*

espansione delle abbreviazioni e rimozione punteggiatura e stopword

*~~In~~ <COUNTRY>~~, the~~ specific chemical requirements laid down ~~by this~~ directive should aim ~~at~~ protecting ~~the~~ health ~~of~~ children ~~from~~ certain substances ~~in~~ toys~~, while the~~ <NUM> environmental concerns presented ~~by~~ toys are addressed ~~by~~ horizontal environmental legislation applying ~~to~~ electrical ~~and~~ electronic toys~~,~~ namely directive <NUMLEX> / European Commission ~~of the~~ **European parliament** ~~and of the~~ council ~~of~~ <DATE>~~.~~*

lemming e lowercase

*<country> specific chemical **requirement lay** directive aim **protect** health **child** certain **substance toy** <num> environmental **concern present toy address** horizontal environmental legislation **apply** electrical electronic **toy** directive <numlex> european commission european parliament council <date>*

divisione in 3-word

- *<country> specific chemical*
- *specific chemical requirement*
- *chemical requirement lay*
- *requirement lay directive*
- *lay directive aim*
- *directive aim protect*
- *aim protect health*
- *protect health child*
- *health child certain*
- *child certain substance*
- *certain substance toy*
- *substance toy <num>*
- *toy <num> environmental*
- *<num> environmental concern*
- *environmental concern present*
- *concern present toy*
- *present toy address*
- *toy address horizontal*
- *address horizontal environmental*
- *horizontal environmental legislation*
- *environmental legislation apply*
- *legislation apply electrical*
- *apply electrical electronic*
- *electrical electronic toy*
- *electronic toy directive*
- *toy directive <numlex>*
- *directive <numlex> european*
- *<numlex> european commission*
- *european commission european*
- *commission european parliament*
- *european parliament council*
- *parliament council <date>*

Definita la pipeline di normalizzazione sono stati, quindi, creati un dataset per ognuna delle parti citate e per ogni elemento, come già detto, è stata applicata una fase di etichettatura con lo scopo di poter risalire al documento di appartenenza. In tal modo, una volta recuperato i frammenti, il cui sistema ritiene i più rilevanti con la query, sarà possibile visionare il documento per intero.

5.1.3 TextPipeline

Prima di iniziare con la descrizione delle tecniche, dobbiamo definire il modo in cui sono state implementate le operazioni di normalizzazione descritte nella sezione precedente. Il modulo responsabile della pipeline di trasformazione è *TextPipeline*: il costruttore prende in input la lingua di riferimento ed un oggetto che è istanza del modulo *spaCy*[2] dell'omonima libreria.

[2]SpaCy è una libreria avanzata per NLP, https://spacy.io/

```
import config
import nltk
class TextPipeline:
    def __init__(self,nlp,lang='en'):
        self.nlp = nlp
        self.stopwords = set(nltk.corpus.stopwords.words(lang))
```

Il campo *stopwords* contiene l'insieme di elementi che identificano parole poco rilevanti. È stata utilizzata la libreria *NLTK*, una delle più utilizzate nel campo *NLP*. Per concentrare tutti i parametri dei vari moduli è stato creato un modulo *config*: in tal modo ogni possibile variabile potrà essere settata attraverso la modifica del modulo in questione. Gli altri metodi della classe *TextPipeline* sono i seguenti:

- **def** convert(self, text, divNGram = True): prende in input il testo da normalizzare e un parametro (*True* di default) che definisce se dividere il testo normalizzato in 3-words; quindi, se *divNGram == True* l'output sarà una lista di 3-words altrimenti una stringa;

- **def** get_last_trigram(self,text): dato il testo in input restituisce l'ultimo 3-word individuato; nel caso in cui non sia stato individuato nessun 3-word, il metodo restituisce la coppia *(None,None)*, altrimenti la coppia *(text,textNorm)*;

- **def** __expand_abbr(self, text): dato il testo in input restituisce il testo con le eventuali abbreviazioni (citate nella sezione precedente) espanse;

- **def** __remove_special_pattern(self,text): dato il testo in input restituisce il testo applicando placeholder sostituitivi ai pattern citati nella sezione precedente.

5.2 Implementazione

La sezione corrente mira a descrivere gli aspetti implementativi delle diverse tecniche proposte nel progetto di tesi. Ogni tecnica deve poter essere interrogata attraverso l'implementazione di un'API il cui compito è quello di ricevere una richiesta sollevata dall'utente e rispondere con gli eventuali elementi più

simili alla query (oggetto della richiesta). Le tecniche implementate sono: *Minhash*, *MRCTA* e *TF-IDF*. Il linguaggio di programmazione utilizzato per la fase di implementazione è *Python* (versione 3.5).

5.2.1 Classe astratta Model

Ogni schema implementa la classe astratta *Model*. Il costruttore di tale classe ha come parametro il tipo di frammento interessato e quindi i possibili valori di *type* sono: *Section, Paragraph, Phrase* e *3Words*. Oltre al settaggio del campo *type*, si crea un'istanza dell'oggetto *spacy* che poi verrà passata al costruttore della classe *TextPipeline*. Il campo *pathDataProc* contiene il percorso del file contenente i frammenti preprocessati.

```
class Model():
    def __init__(self,type):
        nlp = spacy.load('en_core_web_sm')
        self.normalizer = TextPipeline(nlp)
        self.type, self.model = type, None
        self.pathDataProc = config.pathDataProc.format(self.type)
```

Trattandosi di una classe astratta, alcuni metodi non sono implementati questo per poter permettere l'implementazione di ulteriori tecniche e avere, quindi, uno schema comune tra queste. Gli altri metodi sono i seguenti:

- **def** _save(self, path): per il salvataggio del modello;
- **def** load(self): per il caricamento del modello nel campo *self.model*;
- **def** train(self): il corpo di tale metodo dipende fortemente dalla tecnica utilizzata ma in linea generale possiamo dire che verrà prima caricato il file preprocessato, verranno applicati i concetti della relativa tecnica e, infine, verrà salvato il modello su filesystem;
- **def** predict(self, query, threshold, N): data la query testuale chiediamo di restituirci gli N elementi con valore di similarità (con la query) maggiore o uguale al *threshold*;

Il metodo dell'ultimo punto, implica una fase di ranking sui candidati recuperati della tecnica e quindi la classe *Model* dispone di un ulteriore metodo:

```
def __ranking(self,results,query_norm, threshold):
    res_json = []
    if len(results) > 0:
        for item_candidate in results:
            item = metric(query_norm, item_candidate, self.
    normalizer)
            if float(item['lev']) >= threshold:
                res_json += [item]
        res_json = sorted(res_json, key=lambda i: i['lev'],
    reverse=True)
    return res_json
```

Il metodo, il cui codice è mostrato in alto, restituisce un dizionario contenente quegli elementi candidati il cui valore di similarità è maggiore o uguale al valore di *threshold*. La funzione *metric* prende tre parametri: la query normalizzata, il testo dell'elemento candidato non normalizzato e l'oggetto *normalizer* restituendo un dizionario fatto nel seguente modo:

```
{
    'docname': <FilenameDoc>,
    'text': <TestoDelCandidato>,
    'lev': <ValoreDistanzaLevenshtein>
}
```

Ottenuti tutti i valori dei candidati si applica una fase di ordinamento e rimozione di quegli elementi il cui valore del campo *lev* non soddisfa la richiesta. Infine, il dizionario *res_json* viene restituito.

5.2.2 Minhash

Nella seguente sottosezione verranno descritti i punti chiave dell'implementazione relativa alla tecnica LSH e nello specifico quella basata su Minhash. Come detto in 4.3.1 è infattibile computazionalmente applicare anche una sola permutazione di una matrice formata da milioni di righe. Ciò che viene proposto è, dunque, di simulare tale permutazione attraverso l'utilizzo di funzioni hash in modo da mappare elementi (il checksum relativo) in interi ossia in un "indirizzo" che identifica il numero del bucket corrispondente.

Prima di descrivere il processo della generazione della firma dobbiamo porre l'attenzione sulla scelta e la forma delle funzioni hash da scegliere.

Funzioni hash e primo di Mersenne Il primo passo è la definizione di funzioni hash che dato un valore generino l'indirizzo per una posizione nella tabella. Il secondo passo è stabilire un metodo per la risoluzione delle collisioni (situazione in cui la funzione hash applicata a valori diversi generi uno stesso indirizzo nella tabella). Quest'ultima affermazione potrebbe aver creato un po' di confusione, ma ricordo che non stiamo considerando l'intero documento ma le singole componenti di questo. Come abbiamo visto, in linea teorica, nella generazione attraverso minhash, la costruzione della firma è un processo che prende in considerazione un frammento alla volta del documento in input e, in maniera iterativa, genera una firma simile per documenti che condividono frammenti di testo e che quindi saranno stati mappati in bucket uguali rendendoli in tal modo possibili candidati. Un primo requisito per evitare le collisioni è quello di scegliere una dimensione M della tabella molto grande. Anche qui, fare la giusta scelta condiziona le prestazioni. Infatti scegliendo M numero primo le probabilità di ottenere una collisione diminuiscono. Nello specifico sceglieremo come numero il cosiddetto numero primo di Mersenne.

Un numero primo di Mersenne è un numero primo inferiore di uno rispetto a una potenza di due ed esprimibile come: $M_p = 2^p - 1$ con p detto *esponente di Mersenne*. Una condizione necessaria ma non sufficiente per la ricerca di un numero primo di Mersenne è che p sia anch'esso primo. Prendiamo in esame i seguenti tre casi:

1. p non primo $\rightarrow p = 4 \rightarrow M_4 = 2^4 - 1 = 16 - 1 = 15$ che non è primo;
2. p primo $\rightarrow p = 11 \rightarrow M_{11} = 2^{11} - 1 = 2047 = 23 * 89$ quindi non è primo;
3. p primo $\rightarrow p = 7 \rightarrow M_7 = 2^7 - 1 = 127$ è primo.

Se scritti in base binaria, tutti i numeri primi di Mersenne sono primi *repunit*, ovvero sono rappresentati da stringhe di p cifre di valore 1, dove p

è l'esponente primo di Mersenne. Si noti che questa proprietà è posseduta quando si sottrae 1 da tutte le potenze di 2 aventi per esponente un numero primo. Quindi, tutti i candidati a essere numeri primi di Mersenne in base 2 sono primi repunit.

$$(2^p)_2 = 1\underbrace{0\cdots0}_{p}$$
$$M_p = 2^p - 1 \rightarrow (M_p)_2 = \underbrace{1\cdots1}_{p}$$
$$M_2 = 2^2 - 1 \rightarrow (3)_2 = (11)_2$$

Nel codice che andremo a descrivere a breve sono stati utilizzati gli operatori per la manipolazione dei bit per la generazione del numero primo di Mersenne con esponente 61 (M_{61}):

```
_exp_mersenne = 61
_mersenne_prime = (1 << _exp_mersenne) - 1
```

Il simbolo << indica l'operatore di spostamento che applica uno shift di *_exp_mersenne* bit a sinistra identificando cosi una potenza di 2 (2^{61}).

Trovato il numero primo, definiamo la funzione hash da utilizzare per identificare l'indirizzo del bucket dato il relativo checksum del frammento in input. Definiamo una funzione f definita nel seguente modo:

$$f : checksum\_val \rightarrow [0 \cdots M_p]$$

Poichè il valore di f può assumere un valore maggiore della dimensione della tabella hash, abbiamo bisogno di un ulteriore step. Consideriamo la seguente funzione h così definita:

$$h(x,y) = \sum_{n=0}^{b} 2^n (\lfloor \frac{x}{2^n} \rfloor \ mod\ 2)(\lfloor \frac{y}{2^n} \rfloor \ mod\ 2) = x\ AND\ y$$

dove x è il valore in binario del risultato della funzione f e $|x| = |y| = b$. Ignorando la complessità della formula, questa fornisce delle proprietà interessanti: il valore calcolato dalla formula è equivalente all'operazione logica AND, più semplice e veloce da calcolare. Perciò, considerando la funzione $h(x,y) = x\ AND\ y$, h ha le seguenti proprietà:

1. h è simmetrica, $h(x,y) = h(y,x)$

2. $h(x, x) = x$

3. $h(x, y) \leq min(x, y)$

La proprietà numero 3 è quella più interessante; infatti, fissando y uguale alla dimensione massima della tabella hash, quindi al numero di bucket disponibili, possiamo evitare che, per qualche valore di x, tale limite venga superato.

Generazione della firma La creazione della firma avviene in maniera iterativa: se il documento in input è formato da N 3-words verranno applicate N iterazioni per la generazione della relativa firma di lunghezza L. Il costruttore della classe che permette il processo della generazione della firma è *MinhashSignature* il cui costruttore è il seguente:

```
class MinhashSignature():
    def __init__(self, num_perm=128, seed = 1):
        self._max_hash = (1 << 32)
        self._range_hash = self._max_hash - 1
        self.hashfunc = sha1_hash32
        self.seed = seed
        self.hashvalues = np.ones(num_perm, dtype=np.uint64)*
    _max_hash
        generator = np.random.RandomState(seed)
            self.permutations = np.array(
                [(
                    generator.randint(0, _mersenne_prime),
                    generator.randint(1, _mersenne_prime)
                for _ in range(num_perm)
                ], dtype=np.uint64).T
```

Definiti il numero di permutazioni (ossia la lunghezza della firma), la dimensione della tabella hash (*_max_hash*), e il valore che identifica l'indirizzo dell'ultima cella (*_range_hash*), vengono settate alcune funzioni e variabili per la fase di generazione:

- *@self.hashfunc* è la funzione incaricata per il calcolo del checksum del frammento in input;

- *@self.hashvalues* è la colonna trasposta di dimensione *num_perm* della matrice delle firme relativa al documento in questione; poichè la funzione h non produrrà mai un valore maggiore dell'indirizzo dell'ultima cella (*_range_hash*), in ogni cella è contenuto il valore *_max_hash* come riferimento per il valore di infinito;

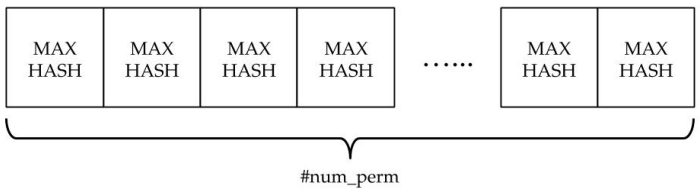


Figura 5.3: stato iniziale vettore *hashvalues*

- *@self.permutations* è la coppia di vettori di lunghezza pari a *num_perm* per la definizione delle altrettante funzioni hash.

Passiamo ora a descrivere le operazioni che permettono la generazione della firma. Ipotizziamo di avere una frase F e di aver estratto i relativi N 3-words f_i con $1 \leq i \leq N$. Per ogni f_i viene chiamato il metodo *update* della classe *MinhashSignature*.

```
def update(self, b):
    checksum_val = self.hashfunc(b)
    rows, c = self.permutations
    f_val = (rows * checksum_val + c) % _mersenne_prime
    h_val = np.bitwise_and(f_val, self._range_hash)
    self.hashvalues = np.minimum(h_val, self.hashvalues)
```

Il valore del checksum è calcolato con la funzione *SHA1*. Tale *checksum_val* è l'input della funzione f che esegue il calcolo delle *num_perm* funzioni hash. Infine, l'ultima operazione permette di individuare i valori minimi tra lo stato della colonna della matrice delle firme (relativa al documento in questione) e il vettore dei valori ottenuti dall'esecuzione delle funzioni hash. Nella figura 5.2.2 è mostrato il processo che porta alla firma finale per testo in input (nota che l'insieme di 3-words è stato ottenuto attraverso una fase di normalizzazione del testo discussa nella sezione 5.1).

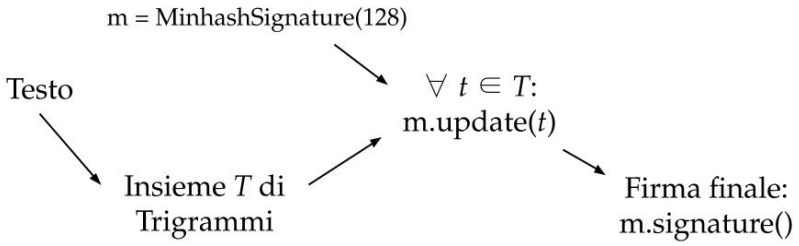


Figura 5.4: Processo generazione firma *Minhash* di un testo in input

Applicazione di Minhash Come detto nella sottosezione 5.2.1 ogni tecnica/schema deve implementare la classe astratta *Model* con i metodi *predict*, *train*, *__save* e *load*. Nel caso di Minhash, il costruttore è così definito:

```
class Minhash(Model):
    def __init__(self,type):
        super().__init__(type)
        self.entryname = 'Minhash'
        self.permutations = config.permutations
        self.path_model = "{}/{}_{}".format(config.path_models,
    self.entryname,self.type)
```

Creato l'oggetto, il metodo *train* prevede il caricamento del dataset (in base al valore del campo *type*) e la chiamata di un metodo privato che gestisce il processo di generazione di tutti gli elementi del dataset.

```
    def train(self):
        with open(self.pathDataProc, 'rb') as handle:
            data = pickle.load(handle)
            index = self.__train_LSH(data)
        self.__save(index,self.path_model)

    def __train_LSH(self,data):
        forest = MinHashLSHForest(num_perm=self.permutations)
        for item in data:
            # per ogni oggetto del dataset genero la firma
            tag, tokens = item['tag'], item['data']
            m = MinhashSignature(num_perm=self.permutations)
            for s in tokens:
                m.update(s.encode('utf8'))
           forest.add(tag,m)
```

```
        forest.index()
        return forest
```

Il metodo *__train_LSH* crea al suo interno una struttura dati che rende possibile la ricerca top-k ossia il recupero dei k risultati con il valore di similarità più alto. Tale struttura è chiamata *LSH_Forest* e verrà descritta più avanti. Ad ogni oggetto *item*, di cui si richiede la firma, viene associato un *tag* che rappresenta il testo originale e non normalizzato di *item*. Il metodo *predict* inizia normalizzando la query (in base al valore di *type* settato al momento della creazione di *Minhash*) per poi passare alla generazioni della firma di quest'ultima.

```
    def predict(self, query, threshold, N):
        # 1. Normalizzo query
        if self.type != 'trigram':
            Trigram = False
            query_norm = self.normalizer.convert(query,divNGram
= False)
            tokens = self.normalizer.convert(query)
        else:
            query, query_norm = self.normalizer.get_last_trigram
(query)
            if query_norm == None:
                return {'query': query,
                        'data': [],
                        'time': '0 ms',
                        'max': N,
                        'time_search': '0 ms',
                        'threshold': threshold
                        }
            Trigram = True
            tokens = [ query_norm ]

        start_time = time.time()
        # Generazione firma per la query
        m = MinhashSignature(num_perm=self.permutations)
        for s in tokens:
            m.update(s.encode('utf8'))
```

Ottenuta la firma si richiama il metodo *query(firma,N)* del modello *LSH_Forest* che restituisce gli N oggetti più simili in base alla stima della distanza di Jaccard. Ottenuti i tag dei candidati viene eseguita la fase di *ranking* per poi produrre un dizionario con le informazioni e risultati della ricerca.

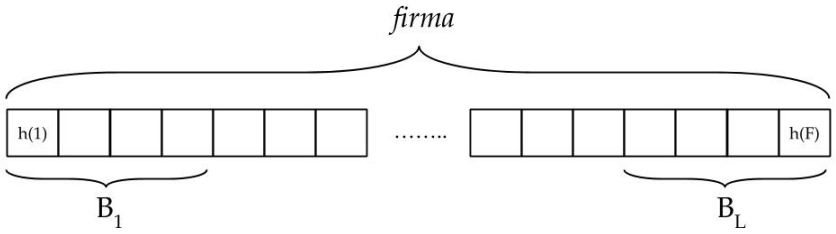
```
        id_candidates = np.array(self.model.query(m, N))
        timing_search = "%.2f ms" % ((time.time() - start_time)
* 1000)
        res_json = self.__ranking(self,id_candidates,query_norm,
threshold)
        timing = "%.2f ms" % ((time.time() - start_time) * 1000)
        return {
            'query': query, 'data': res_json,
            'time': timing, 'max': N,
            'time_search': timing_search,
            'threshold': threshold, 'algoritm': self.entryname
        }
```

LSH Forest Dato un dataset di piccole dimensioni, si potrebbe semplicemente confrontare le firme a coppie per la ricerca dei candidati. Tuttavia, un approccio di questo genere diventa impraticabile a causa dei costi lineari anche con dataset di media dimensione. La soluzione è sviluppare un meccanismo di indicizzazione che possa, a *query time*, prendere in esame solo un piccolo sottoinsieme di elementi per poi applicare la fase di ranking e confronto. Diverse sono le proprietà che un *indexer* deve seguire:

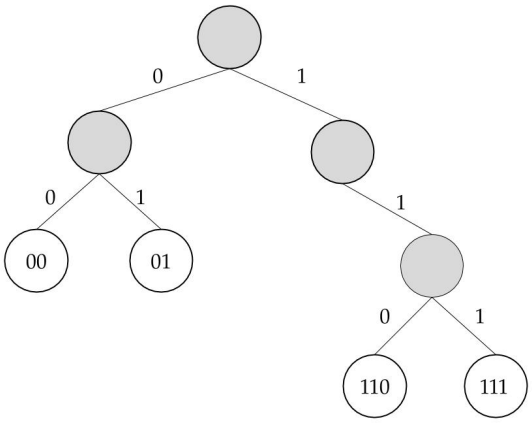
- *Accuratezza*, l'insieme dei candidati recuperati dall'indexer deve contenere gli elementi più simili alla query;
- *Efficienza rispetto la query*, il numero dei candidati deve essere il più piccolo possibile, per ridurre il numero di operazione I/O e i costi dei confronti;
- *Efficienza rispetto la manutenzione*, l'indice deve essere costruito con una singola passata del dataset e la cancellazione o inserimento di un indice deve essere efficiente;

- *Indipendente dal dominio*, l'indice non dovrebbe richiede alcuno sforzo aggiuntivo per il tuning dei parametri per diversi dataset di partenza;
- *Spazio minimo*, l'indice dovrebbe utilizzare il minor spazio possibile, idealmente deve essere lineare alla dimensione del dataset.

In [BCG05] è presentato *LSH Forest* ossia un indice per l'*approximate similarity search* che affronta e risolve i problemi elencati precedentemente. LSH Forest migliora il problema dell'accuratezza ed elimina il bisogno di applicare una fase di tuning dei parametri e il suo funzionamento si basa sullo schema LSH. Ricordando che lo schema LSH genera una firma di dimensione F prefissata (il valore di F indica il numero di funzioni hash scelte) in modo che $sign(item) = (h_1(item), \cdots, h_F(item))$, possiamo considerare tale firma composta da L blocchi di $\frac{|F|}{L} = k$ dimensione ciascuno.



Possiamo cosi costruire un albero di ricerca (*prefix tree*) per l'insieme di tutte le firme in cui ogni foglia corrisponde ad un punto e chiamiamo questo *LSH Tree*. Una collezione di L LSH Tree è chiamata LSH Forest. La figura 5.2.2 mostra un LSH Tree contenente 4 punti le cui hash function producono un bit di output. Le foglie corrispondono ai 4 punti con le relative etichette mentre i cerchi grigi sono i nodi interni dell'albero. È importante osservare che il valore nelle foglie rappresenta il percorso dalla radice a una di queste. Inoltre, non è necessario che tutti i nodi interni abbiano due figli



A livello implementativo, la dichiarazione dell'oggetto *LSHForest* implica la definizione di due parametri: *num_perm* e *l*, rispettivamente la lunghezza della firma per ogni punto della collezione e il numero di LSH Tree desiderati.

```
class MinHashLSHForest(object):
    def __init__(self, num_perm=128, l=8):
        if l <= 0 or num_perm <= 0:
            raise ValueError("num_perm e l devono essere
    positivi")
        if l > num_perm:
            raise ValueError("Valore di l non permesso!")
        self.l = l
        self.k = int(num_perm / l)
        self.hashtables = [ {} for _ in range(self.l)]
        self.hashranges = [(i*self.k, (i+1)*self.k) for i in
    range(self.l)]
        self.keys =  {}
        self.sorted_hashtables = [[] for _ in range(self.l)]
```

Come già detto precedentemente, il valore dei parametri *num_perm* e *l* determinano la dimensione dei blocchi: infatti, considerando il caso in questione, avremo per ogni firma una divisione in 8 blocchi contenente ognuno 16 elementi. Gli altri campi sono utilizzati nella fase di aggiunta, indicizzazione e ricerca:

- *@hashtables*, array di dizionari, tanti quanti il numero di LSH Tree prefissati;

- *@hashranges*, array che definisce i limiti inferiori e superiori di ogni partizione;
- *@keys*, come vedremo nel metodo che permette l'inserimento, ogni partizione (una lista di interi) può essere mappata in un oggetto di tipo *Byte* che verrà poi utilizzato nella fase di ricerca;
- *@sorted_hashtables*, per ogni LSH Tree verrà utilizzato tale campo per effettuare la ricerca.

Il metodo *add()* prende in input due parametri: il primo è la label con cui identificare il testo la cui rappresentazione (il secondo campo) è data dalla firma (un oggetto Minhash, definito precedentemente).

```
def add(self, key, signature):
    if len(signature) < self.k*self.l:
        raise ValueError("Lunghezza firma errata")
    if key in self.keys:
        raise ValueError("Esiste gia' un elemento con questa
chiave")
    self.keys[key] = [self._H(signature.hashvalues[start:end
])
            for start, end in self.hashranges]
    for H, hashtable in zip(self.keys[key], self.hashtables):
        hashtable[H].append(key)

def _H(self, hs):
    return bytes(hs.byteswap().data)
```

Dopo aver effettuato i controlli sulla lunghezza della firma e sull'unicità della label, le l partizioni vengono mappate in un oggetto *Byte* utilizzato per indicizzare la posizione della cella nella hashtable relativa. Infine ogni i-esima hashtable viene popolata con l'i-esima partizione e nello specifico: ogni valore ottenuto dalla funzione *_H()* diventa una chiave della relative tabella che punterà alla lista di etichette che contengono quella esatta porzione. Per poter permettere una ricerca si ha la necessità di ordinare le chiavi all'interno di ogni hashtable ed il metodo che consente ciò è *index()*.

```
def index(self):
for i, hashtable in enumerate(self.hashtables):
```

```
            self.sorted_hashtables[i] = [H for H in hashtable.
    keys()]
            self.sorted_hashtables[i].sort()
```

Per la fase di ricerca, consideriamo L prefix tree costruiti con i metodi precedenti. Una query per la ricerca degli m nearest neighbors è applicata attraversando gli LSH Trees in due fasi:

1. *top-down*, per ogni LSH Tree T_i si cerca la foglia che ha il più lungo prefisso con la query q e sia $x = max_i^L\{x_i\}$ il massimo livello tra le foglie trovate per gli L LSH Tree;

2. *bottom-up*, collezioniamo un insieme $\mathbb{M}$ punti da LSH Forest partendo dal livello x verso la radice per ogni T_i.

Ottenuto $\mathbb{M}$ viene applicata una fase di ordinamento in modo da assicurare una lista distinta di elementi che che soddisfino la similarità con la query. Il metodo *query* prende in input due parametri: la firma e il numero massimo di risultati desiderati.

```
def query(self, signature, n_res):
    if len(signature) < self.k*self.l:
        raise ValueError("Lunghezza firma errata!")
    results = set()
    r = self.k
    while r > 0:
        for key in self._query(signature, r):
            results.add(key)
            if len(results) >= n_res:
                return list(results)
        r -= 1
    return list(results)
```

Il metodo inizia considerando ogni B_i per intero (r=k). Se il numero di elementi richiesti è stato raggiunto (len(results) == n_res) non viene decrementato nulla e viene restituita la lista con i candidati; altrimenti viene decrementato il valore di r andando cosi a considerare le sotto porzioni di ogni blocco. Le fasi citate precedentemente le troviamo implementate nel metodo privato *__query*.

```
def _query(self, minhash, r):
    hps = [self._H(minhash.hashvalues[start:start+r])
           for start, _ in self.hashranges]
    prefix_size = len(hps[0])
    for ht, hp, hashtable in zip(self.sorted_hashtables, hps,
self.hashtables):
        # top-down
        x = self._binary_search(len(ht),
            lambda x : ht[x][:prefix_size] >= hp)
        # bottom-up
        if x < len(ht) and ht[x][:prefix_size] == hp:
            j = x
            while j < len(ht) and ht[j][:prefix_size] == hp:
                for key in hashtable[ht[j]]:
                    yield key
                j += 1
```

Come già detto, r è inizialmente la dimensione totale della porzione considerate in *hps*. Dopo avere calcolato i relativi indirizzi nelle varie hashtable, inizia la ricerca della fase *top-down*. Attraverso una ricerca binaria viene individuato il valore di x e successivamente si risale alla ricerca di altri elementi (fase bottom-up).

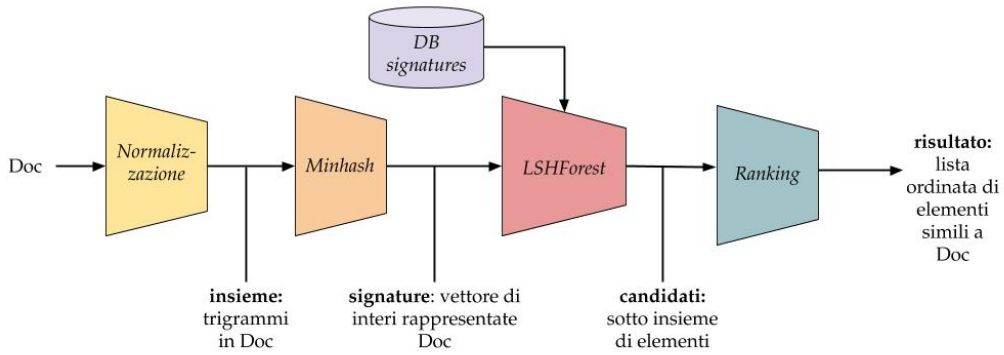


Figura 5.5: Utilizzo di Minhash per il recupero dei documenti simili

In figura 5.5 è mostrato il processo per la ricerca di documenti simili attraverso l'utilizzo di Minhash, ma tale pipeline è comunque applicabile a qualunque altra tecnica descritta in questa dissertazione. La fase di *preprocessing* viene utilizzata per generare le firme dei documenti della collezione e

inserirle in un database. La fase di ricerca inizia con la fase di normalizzazione al documento in input producendo un insieme di 3-words normalizzati. A questo punto è possibile generare la firma partendo da questo insieme e tramite la firma ottenuta individuare i candidati. Infine, alla lista dei candidati viene applicata un'ultima fase, quella di ranking, il cui obiettivo è produrre una lista ordinata di elementi in base alla distanza di jaccard di cui Minhash si basa.

Complessità La complessità per la generazione della firma è $O(|D|)$ ossia lineare in base al numero di shingles all'interno del documento D. Il tempo di ricerca con l'algoritmo *LSHForest* descritto dipende dal numero di hash tables L considerate (un numero maggiore di hash tables determina una maggiore accuratezza) ed è quindi $O(L \log n)$ con n numero di elementi nella collezione. Se L è costante possiamo indicare la complessità con $O(\log n)$. La fase di preproccessing implica il dover generare le firme per ogni documento e l'inserimento nelle L hash tables: indicando con d la dimensione media di ogni documento della collezione, la complessità è $O(nd) + O(nL)$.

5.2.3 Term Frequency-Inverse Document Frequency

Le ultime due implementazini riguardano concetti visti nella sezione 3.1 e nello specifico la metodologia TF-IDF e una sua ottimizzazione MRCTA (sezione 4.4).

L'implementazione di questo algoritmo è stata fatta utilizzando la libreria *scikit-learn* di Python. Come visto nella sezione relativa a TF-IDF, l'obiettivo è assegnare ad ogni token uno score che ne identifica il peso in base alla collezione dei documenti (parole rare avranno score più alto di parole utilizzate con maggiore frequenza). Nel costruttore di *TextVectotizer*, dopo aver creato l'oggetto *TfidfVectorizer* della libreria *sklearn*, vengono definiti i path per il salvataggio di quest'ultimo e del classificatore knn. Per la fase di ricerca è stato utilizzato un algorimo di machine learning supervisionato, *k-nn*, che permette di individuare in uno spazio dimensionale i k punti più vicini al punto della query (sezione 4).

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
from sklearn.neighbors import NearestNeighbors
import config
class TextVectorizer(Model):
    def __init__(self, type):
        super().__init__(type)
        self.entryname = 'TF-IDF'
        with open(self.pathDataProc, 'rb') as handle:
            self.data = pickle.load(handle)
        self.labels = [item['tag'] for item in self.data]
        self.knn = None
        self.vectorizer = TfidfVectorizer(ngram_range=(3,3))
        # path dei modelli
        self.path_knn = <path_classificatore>
        self.path_vec = <path_vectorizer>

    def save(self):
        # save self.knn, self.vectrorizer
        ...
    def load(self):
        self.knn = ...
        self.vectorizer = ...
```

Caricati gli oggetti normalizzati, vengono generate le "firme" dei primi con il metodo TF-IDF. Il metodo *fit_transform()* prende in input una lista e produce i relativi vettori degli elementi in input. Il risultato verrà poi dato in input al classificatore che definirà successivamente i 10 (k=10) punti più vicini per un determinato *query point* in base alla *cosine distance*.

```
    def train(self):
        data_train = [ item['normalized'] for item in data]
        matrix = self.vectorizer.fit_transform(data_train)
        self.knn = NearestNeighbors(n_neighbors=10, metric='
    cosine').fit(matrix)
        self.save()
```

Infine per il metodo *predict()* viene applicato il solito schema: viene creata la rappresentazione vettoriale del testo in input, trovati gli id dei punti più vicini, se ne recuperano le label su cui verrà poi applicata la fase di riordinamento del metodo *__ranking()*.

```
    def predict(self, query, threshold, N):
```

```
        start_time = time.time()
        # 1. Normalizzo query
        query_norm = self.normalizer.convert(query, divNGram=
    False)
        # 2. Genero vettore TF-IDF
        query_vec = self.vectorizer.transform([query_norm])
        # 3. Lista degli N id dei punti vicini
        dists, id_candidates = self.knn.kneighbors(query_vec)
        timing_search = "%.2f ms" % ((time.time() - start_time)
    * 1000)
        # 4. recupero delle label degli id ottenuti dalla
    ricerca
        labels = []
        for id in id_candidates[0]:
                labels += [self.labels[id]]
        # 5. Ordinamento per distanza di Lev
        res_json = self.__ranking(self,labels,query_norm,
    threshold)
        timing = "%.2f ms" % ((time.time() - start_time) * 1000)
        return {
            'query': query, 'data': res_json,
            'time': timing, 'max': N,
            'time_search': timing_search,
            'threshold': threshold, 'algoritm': self.entryname
        }
```

5.2.4 Multi-Reference Cosine Text Algorithm

Per l'implementazione di questa tecnica dobbiamo focalizzare l'attenzione sul modo in cui viene generato il testo riferimento. Come detto nella sezione 4.4, MRCTA è una tecnica che permette di ridurre il numero di dimensioni del vettore relativo il testo in input. Infatti applicando TF-IDF e avendo 10^6 token otterremo, un vettore di dimensione 10^6 per ogni testo che si vuole rappresentare. Il *reftext* è stato generato tenendo in considerazione gli N 3-words con lo score TF-IDF più elevato. Partiamo con l'analisi del costruttore dell'oggetto *MRCTA:*

```
import config
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
from sklearn.neighbors import NearestNeighbors
class MRCTA(Model):
    def __init__(self,type, sign = 256):
        super().__init__(type)
        self.entryname = 'MRCTA'
        self.lenghtSign = sign # lunghezza firma
        self.wordsinpart = 20  # num elementi per ogni
    partizione
        # numero totali di elementi con score piu alto da
    considerare
        self.numHigherScoreWord = self.lenghtSign * self.
    wordsinpart

        with open(self.pathDataProc, 'rb') as f:
            self.data = pickle.load(f)
        self.labels = [item['tag'] for item in self.data]

        self.vectorizer, self.knn = None
        self.ref_text, self.ref_text_vec = None

        # paths
        self.path_ref_text = config.path_reftext + "_" + self.
    type
        self.path_model_vec = config.path_model_vectorizer + "_"
     + self.type
        self.path_model_knn = config.path_model_knn + "_" + self
    .type
```

Definita la lunghezza della firma desiderata e la dimensione di ogni porzione che divide il *reftext* si procede con il determinare il numero di parole con score TF-IDF più alto. La fase di train, invece, prevede la creazione del *reftext* mediante il metodo *gen_ref_text()* che prende in input un intero N ossia creerà una stringa formata da *numHigherScoreWord* con score più alto.

```
    def train(self):
        # 1. generazione del ref text
        ref_text = self.gen_ref_text(N=self.numHigherScoreWord)
        # 2. Allenamento di TF-IDF su tutta la collezione
        corpus = [d['normalized'] for d in self.data]
        self.vectorizer = TfidfVectorizer()
```

```
        self.vectorizer.fit(corpus)
        # 3. Creazione dei lenghtSign vettori ognuno dei quali
    appartenente ad una porzione del ref text
        self.ref_text_vec = self.__ref_text_vector(ref_text)
        signs = [self.gen_sign(item) item in corpus]
        # 4. Training di knn
        self.knn = NearestNeighbors(n_neighbors=10,metric='
    cosine').fit(signs)
        # salvataggio knn, vectorizer e ref_text
        self.save_model()
```

Ottenuto il *reftext* si procede con la creazione delle firme TF-IDF (che andremo a comprimere) su tutta la collezione. Calcolata la matrice TF-IDF viene partizionato *reftext* in un numero di parti pari alla lunghezza della firma desiderata (*self.lenghtSign*) e ogni parte viene vettorizzata con *self.vectorizer*.

```
    def __ref_text_vector(self, ref_text):
        ref_text = ref_text.split()
        # divisione in lenghtSign partizioni di dimensione
    wordsinparts
        ranges = [
            (i,(i+1)*self.wordsinpart)
            for i in range(0,self.lenghtSign)]
        parts = [
            " ".join(ref_text[start, end])
            for start, end in ranges
            ]
        # vettorizzazione delle parti estratte
        vector_parts = [
            self.vectorizer.transform([i]).toarray()[0]
            for i in parts
        ]
        return vector_parts
```

La fase di ricerca è uguale a quella descritta precedentemente per il metodo *predict* di TF-IDF ma l'unica operazione diffente è naturalmente la generazione del vettore della query in input.

```
    def predict(self, query, threshold, N):
        # 1. Normalizzo query
            ...
```

```
        # 2. Genero vettore MRCTA
        query_vec = self.gen_sign(query_norm)
            ...
        return {...}
```

Quindi, passato in input il testo da rappresentare e calcolato il relativo vettore TF-IDF si procede con il calcolo della *cosine distance* con ognuna delle 256 (dimensione della firma desiderata) partizioni.

```
    def gen_sign(self, text):
        sign = []
        text_vec = self.vectorizer.transform([text]).toarray()
[0]
        for part in self.ref_text_vec:
            sign += [self.__cosine_similarity(part, text_vec)]
        return sign
```

Fino ad ora è stata descritta la pipeline di normalizzazione e le varie implementazioni delle tecniche analizzate nei capitoli 3 e 4 ponendo maggiore enfasi su una particolare istanza di LSH ossia Minhash. Sono stati visti, inoltre, i metodi che ogni classe, istanza di una tecnica, deve possedere per interagire con il sistema:

- un costruttore che definisce la dimensione della rappresentazione del testo con i relativi parametri;
- metodi per il salvataggio e il caricamento del modello;
- il metodo *train()* per processare una collezione di partenza secondo le regole della tecnica;
- il metodo *predict()* per ottenere gli elementi che secondo il sistema sono i piu' simili con la query in input tenendo conte della soglia e del massimo numero di risultati.

5.3 Funzionalità e utilizzo tramite API

Per renderne accessibile l'utilizzo è stata implementata, per ogni tecnica, un'API permettendo, quindi, l'interrogazione e l'integrazione all'interno dell'ambiente SHE. Ogni tecnica, perciò, deve implementare un'API fornendo i metodi HTTP mostrati nella tabella seguente:

Metodo HTTP	**URI**	**Azione**
GET	http://[hostname]/	Fornisce informazioni sulla tecnica
GET	http://[hostname]/connect	Caricamento dei file di configurazione della tecnica
POST	http://[hostname]/query	Interrogazione in base al tipo indicato nel body

Tabella 5.1: Metodi HTTP delle API

Le API sono state implementate utilizzando il framework Python *Flask*. Tale framework permette la creazione, in maniera molte semplice, di API. Infatti, per creare una *route* (un URI per un'azione) si procede nel seguente modo:

```
@app.route('/', methods=['GET'])
def index():
        response = app.response_class(
                response=json.dumps({'data': "Welcome Minhash
    Entrypoint!"}, indent=4),
                status=200,
                mimetype='application/json'
        )
        return response
```

Qui è stata definita una route per il metodo GET fornendo quindi l'implementazione descritta nella prima riga della tabella 5.1 per la tecnica Minhash. La route relativa al caricamento dei file di configurazione del modello avrà al suo interno una chiamata al metodo *load* per ogni tipo che la tecnica può gestire. Ad esempio, Minhash permette la gestione di tutte e 4 le tipolo-

gie e quindi al momento della chiamata alla routa *GET /connect* verranno istanziat i 4 oggetti Minhash.

```
@app.route('connect/', methods=['GET'])
def connect():
        Minhash_f, Minhash_p = Minhash('phrase'), Minhash('
    paragraph')
        Minhash_s, Minhash_t = Minhash('section'), Minhash('
    trigram')

        Minhash_f.load()
        Minhash_p.load()
        Minhash_s.load()
        Minhash_t.load()

        return app.response_class(
                response=json.dumps({
                        'data': 'Success',
                        'path':config.path_models,
                        'ip': config.ip
                }, indent=4, sort_keys=True), status=200,
                mimetype='application/json'
        )
```

Per la route di interrogazione *POST /query* dobbiamo definire prima il body del messaggio di richiesta e di risposta.

Richiesta Per effettuare un'interrogazione all'API delle tecnica, il relativo messaggio di richiesta deve avere la seguente struttura:

- **data**, la parte di testo per cui si vogliono reperire gli elementi piú simili;
- **type**, una stringa che definisce il tipo di ricerca che si vuole effettuare (Section, Paragraph, Phrase e in alcuni casi 3Words);
- **max**, numero massimo di risultati richiesti;
- **threshold**, valore minimo di similarità che i risultati devono avere con il testo del campo *data* ($threshold \in [0, 1]$);

Ad esempio, nel json seguente si stanno richiedendo, per il testo in data, i risultati relativi a frammenti di tipo 'Phrase' e tali risultati devono avere similarità maggiore o uguale a 0.3 con la query:

```
{
    'data': 'The European Commission, having regard to the
   treaty on the functioning of the European union',
    'type': 'Phrase',
    'max': 10,
    'threshold': 0.3
}
```

Risposta Inviata il messaggio di interrogazione all'API corrispondente, questa, in base al valore nel campo *type*, chiama il campo *predict* e, dopo la fase di riordinamento discussa nella sottosezione 5.2.1, restituisce un messaggio di risposta con i seguenti campi:

- **query**, il testo per soggetto dell'interrogazione;
- **data**, lista dei risultati forniti in cui ogni oggetto ha a sua volta una struttura che contiene il documento di appartenenza, il testo dell'oggetto restituito ed il grado di similarità con la query e in base alla distanza di Levenshtein;
- **time**, tempo totale di risposta, dalla normalizzazione della query fino al termine della fase di ranking;
- **time_search**, tempo impiegato per la ricerca (normalizzazione+tempo di ricerca)
- **max**, numero massimo di risultati richiesti;
- **threshold**, valore di soglia richiesto;
- **algorithm**, nome dell'entrypoint interrogato.

Di seguito è mostrato un possibile messaggio di risposta alla richiesta descritta precedentemente:

```
{
        'query': 'The European Commission, having regard to
    the treaty on the functioning of the European union',
        data: [{
                'docname': '32012D0160.html',
                'lev': 0.86,
                'text': '2012/c 198/06 the european
    commission, having regard to the treaty on the functioning
     of the european union, whereas'
        },
        ... ],
        'time': '169.28 ms',
        'time_search: '5.07 ms',
        'max': 10,
        'threshold': 0.3,
        'algorithm': 'Minhash'
}
```

5.4 SHE: Interface

In questa sezione verrà descritto l'ambiente *SHE* (*Similarity Hashing Enviroment*) il cui scopo è quello di fornire un meccanismo per il confronto di varie tecniche di similarità. Come discusso nella sezione precedente, ogni tecnica deve poter essere accessibile mediante un URL che identifica un entrypoint. Seguendo gli standard definiti, come i messaggi di richiesta e risposta, qualsiasi tecnica può partecipare alla fase di confronto che verrà poi utilizzata per la fase di testing (aspetto che verrà trattato e discusso nel capitolo successivo).

L'interfaccia web che verrà presentata è stata implementata con l'utilizzo dei framework *Bootstrap* (view) e *JQuery* (controller, business logic).

Definiti gli entrypoints per ogni tecnica soggetta al testing, queste vengono integrate nel sistema attraverso il file *entrypoints.js*. In questo file sono presenti le strutture dati che ne permettono l'integrazione nell'ambiente e, per ogni meccanismo di ricerca, sono richiesti: un nome identificativo, URL dell'API e una breve descrizione della tecnica in questione. Inserite le informazioni nel file è possibile visualizzare la lista delle tecniche mediante la

sezione *Entries* dell'interfaccia web ove verranno mostrati gli identificativi e gli entrypoints delle tecniche disponibili. L'ambiente che verrà descritto è composto da due parti: una rivolta alla sviluppatore (*Dev Interface*[3]) e l'altra al tester (*Test Interface*[4]).

5.4.1 Dev Interface

L'interfaccia di sviluppo ha il compito di testare i requisiti implementativi che l'API della tecnica deve soddisfare: a questo livello lo sviluppatore può interagire con le funzionalità fornite dall'interfaccia di sviluppo per testare il comportamento del sistema da lui implementato, la gestione delle richieste e delle risposte, i tempi necessari per la ricerca e i risultati con i relativi valori di similarità. Questo fornisce al programmatore un modo per mettere il suo lavoro a confronto con altre tecniche e, sulla base di questi confronti, apportare eventuali migliorie al proprio sistema.

L'interfaccia mette a disposizione un meccanismo che consente l'invio simulataneo di una richiesta a tutti gli entrypoints disponibili. La richiesta è formata dal valore della *textarea* presente nella view *Dashboard*, e dei valori di *threshold*, tipo di richiesta e numero di risultati. Ogni campo del messaggio può essere modificato attraverso la *Settings* accessibile dalla view *Entries*.

Con l'arrivo delle risposte (secondo lo standard definito) da parte delle API, ognuna di queste andrà a popolare il relativo pannello. Ogni pannello è identificato dal nome della tecnica dichiarata nel file precedentemente definito. Al momento del caricamento dell'interfaccia, verranno creati N pannelli tanti quanti sono gli entrypoints dichiarati. Ogni pannello è posto al di sotto delle textarea della view *Dashboard* e ognuno di questi conterrà i risultati ottenuti dall'interrogazione dell'API relativa.

Ad esempio, nella figura 5.6 è mostrato il pannello relativo la risposta (dopo una richiesta di tipo *Phrase*) da parte della tecnica con identificativo *Minhash*. Per rendere subito evidente la qualità della risposta, sono stati scelti 5 colori che identificano una classe di similarità per ogni frase suggerita.

[3]http://site181928.tw.cs.unibo.it/she/
[4]http://site181928.tw.cs.unibo.it/she/test

Figura 5.6: Pannello risultati per la tecnica Minhash

Ogni range definisce un colore dove rosso indica un cattivo suggerimento e verde un ottimo suggerimento.

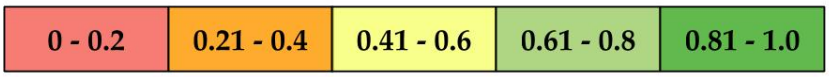


Figura 5.7: Palette di colori per definire la classe di similarità

Inoltre, le differenze testuali tra la query e l'elemento suggerito sono state evidenziate con due colori: il rosso per indicare le parti della query non condivise con l'i-esimo risultato; il verde per indicare le parti del risultato mancanti nella query.

5.4.2 Test Interface

L'interfaccia di test, invece, ha l'obiettivo di individuare attraverso il suo utilizzo qual è la tecnica che meglio raggiunge l'obiettivo di questo elaborato. Come vedremo nel capitolo successivo, saranno considerati solo due delle quattro tecniche viste nel corso dei capitoli precedenti questo per evitare di appesantire il tester con troppi risultati o scelte provenienti dalle varie tecniche. L'architettura è la medesima vista con l'interfaccia Env con l'aggiunta

di alcuni componenti che permettono di catturare le informazioni utili per la fase di valutazione.

Definite le tecniche soggette al test, i vari task da sottoporre ai tester sono inseriti nel file json *tasks.json* avente n oggetti tanti quanti sono i task. Ogni task è composto dai seguenti campi:

- *type*, tipologia del task (*Paragraph*, *Section*, *Phrase*) che verrà utilizzato dal sistema per interrogare le varie tecniche;
- *task*, obiettivo che l'utente deve raggiungere;
- *init*, frammento iniziale da cui l'utente parte per raggiungere l'obiettivo descritto nel campo *taks*.

Ad esempio, il frammento di json seguente descrive un task posto all'utente in cui gli è stato chiesto di redigere un paragrafo che parla di un qualche argomento citando gli articoli indicati e partendo dal frammento posto in *init*.

```
[
   {
        'type': 'Paragraph',
        'task': 'Please draft a paragraph stating that
   supervisory authorities may adopt provisional measures to
   protect the rights of threatened subjects, by derogating
   to articles 21, 22 and 23 of this regulation, for a period
    of validity up to three months,by adopting temporary
   measures, provided that such measures are communicated
   without delay to the supervisory authority as well as the
   reason they were adopted.',
        'init': 'In exceptional circumstances, where a
   supervisory authority concerned considers that there is an
    urgent need to act in order to protect the rights of
   threatened subjects, it may'
    }, { ... }
]
```

In questa parte della dissertazione abbiamo discusso delle varie implementazioni trattate nei capitoli precedenti focalizzando maggiormente l'attenzione sulle implementazioni della tecniche *Minhash* e *MRCTA*. Si è discusso anche di aspetti riguardanti la normalizzazione del testo che permette di ottenere dei risultati migliori andando a rimuovere alcune parti del testo considerate superflue (le stopwords) o andando a riduerre i verbi alla loro forma all'infinito. Sono state descritte, inoltre, due interfacce web di cui la prima *Env Interface* permette il confronto, per chi sviluppa una nuova tecnica seguendo lo standard definito, in modo da assicurarsi del corretto funzionamento; mentre la seconda *Test Interface* è destinata alla fase di testing il cui utente principale è un utente esperto del dominio della collezione dei documenti di partenza. Quest'ultimo argomento sarà trattato nel capitolo successivo in cui andremo a discutere dei test effettuati e delle relative valutazioni.

Capitolo 6

Test e Valutazioni

Nei capitoli precedenti sono stati presentati gli aspetti utili ed essenziali del progetto di tesi. È stato descritto, sia a livello teorico che pratico, come l'utilizzo di tecniche basate sul calcolo della similarità possa aiutare al recupero di documenti in modo da facilitare la fase di editing di questi. Sono state presentate, inoltre, le due interfacce che permettono l'interfacciarsi con ognuna delle tecniche messe a disposizione sia per la fase di sviluppo che di testing. Di seguito, avendo parlato di tutte le componenti che definisco struttura del progetto di tesi, saranno presentati i risultati ottenuti dalla fase di test. Infine saranno presentate le valutazioni finali che porteranno a una serie di proposte di sviluppi futuri le quali saranno elencate nell'ultimo capitolo.

6.1 Struttura del test

Come detto durante l'introduzione, il test mira a valutare la soddisfazione dell'utente nell'utilizzo del sistema. Attraverso questa fase si cercherà di rilevare quale tecnica è stata giudicata più utile dagli utenti del test. Verranno presentati sei task che descrivono l'attività da completare. Queste sono definite, come detto nel capitolo precedente, da un frammento di testo iniziale che viene mostrato all'utente il quale dovrà continuare a digitare seguendo le indicazioni fornite dalla descrizione del task. Durante la fase di scrittura nella textarea dell'interfaccia di test verranno mostrati all'utente i suggerimenti provenienti dalle tre tecniche considerate ossia: Minhash, MRCTA e TLSH [GC19]. Su ogni elemento prodotto dalle tecniche l'utente può decidere se utilizzarlo o meno per velocizzare la scrittura e il completamento del task. Ad esempio, nella figura 6.1 è mostrato uno screen di un suggerimento prodotto e attraverso bottone *Add* è possibile includere il frammento considerato il quale verrà concatenato con il testo contenuto nella textarea e quindi pronto per essere modificato al fine di raggiungere il task.

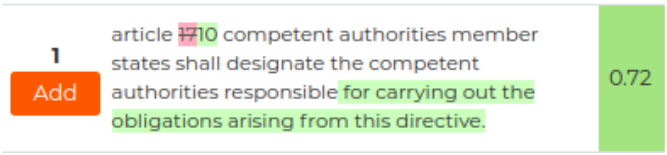


Figura 6.1: Suggerimento di uno dei sistemi

Alla fine di ogni task, verrà chiesto all'utente di fornire una valutazione sui sistemi. Questo viene fatto chiedendo all'utente di distribuire 12 punti tra le varie opzioni. Ad esempio, l'utente può decidere di assegnare tutti e 12 i punti all'opzione *Manual* il che significa che l'utente ha ritenuto non rilevanti i risultati prodotti dalle tre tecniche e che ha preferito completare il task in modo manuale ossia senza l'aiuto di nessun supporto; oppure può decidere di distribuire i 12 punti in base all'andamento del task: in figura 6.2 è mostrato lo stato in cui l'utente abbia preferito l'utilizzo della tecnica Minhash rispetto alle altre opzioni.

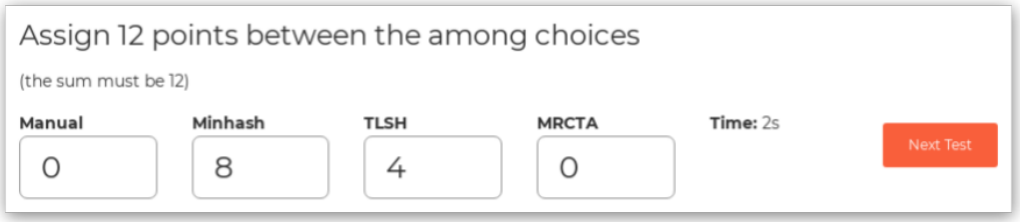


Figura 6.2: Pannello per l'assegnamento dei punti relativi ad un task

Finiti i task verrà chiesto all'utente di compilare un questionario sulla base dei punteggi forniti alla fine di ogni task. Il questionario verrà applicato alla sola tecnica con il punteggio più alto. Se fossero presenti tecniche con lo stesso punteggio il questionario viene effettuato per tutte le tecniche a disposizione mentre nessun questionario sarà chiesto nel caso in cui l'opzione *Manual* abbia avuto il punteggio più elevato. Oltre l'eventuale compilazione del/dei questionario/i verrà chiesto all'utente di scrivere la propria opinione riguardo l'andamento del test in generale.

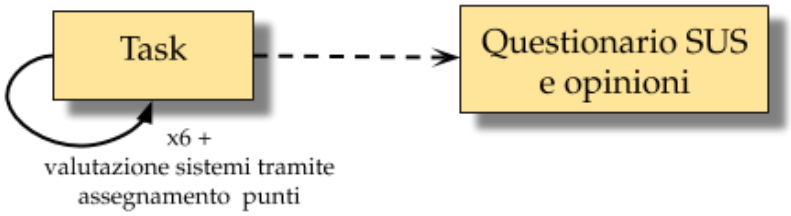


Figura 6.3: Struttura del test

6.1.1 Questionario SUS

Il questionario SUS [LS09] (System Usability Scale) 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. Il questionario è composto da 10 punti, che si alternano tra affermazioni positive (dispari) e negative (pari). Quando viene compilato, ai partecipanti viene richiesto di assegnare un punteggio alle seguenti 10 affermazioni indi-

cando per ognuna un punteggio che va da 1 (fortemente in disaccordo) a 5 (fortemente in accordo):

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.

Ottenuti i punteggi per ogni affermazione, il calcolo dei risultati SUS avviene attraverso il seguente processo:

1. per le affermazioni positive (dispari) si sottrae 1 dal punteggio ($score - 1$);
2. per le affermazioni negative (pari) si sottrae il punteggio da 5 ($5 - score$);
3. i valori vengono sommati e moltiplicati per 2.5 in modo da ottenere un valore $final\_score \in [0, 100]$;
4. infine, ottenuti tutti i $final\_score$ relativi ai questionari di ogni utente se ne calcola la media.

. Un punteggio SUS maggiore o uguale a 68 indica una buona usabilità del sistema. Come detto nella fase introduttiva del corrente elaborato, l'obiettivo principale del lavoro di tesi non è solo quello di progettare un sistema di suggerimento che sia efficiente in termini di tempo e precisione ma anche di ottenere una soddisfazione da parte dell'utente utilizzatore del sistema e quindi porre le basi per eventuali sviluppi futuri. Nell'analisi soggettiva, la cui struttura è stata descritta precedentemente e i cui risultati verranno trattati nella sezione successiva, l'attenzione è spostata totalmente sull'esperienza dell'utente.

6.1.2 Descrizione dei tester

I partecipanti al test possono essere considerati esperti del dominio infatti, data la tematica della collezione di partenza, si è pensato di coinvolgere degli studenti di giurisprudenza dell'*University of Lapland* (Rovaniemi, Finlandia). In totale gli studenti sono stati 23 e possiamo dividerli in due gruppi: il primo composto da 7 studenti, tra il primo e il secondo anno, frequentati del corso di *Introduzione all'informatica giuridica* tenuto dal prof. Dino Girardi; mentre il secondo gruppo composto da 16 studenti del quarto anno e frequentanti del corso di *Privacy Online* del prof. Aleksander Wiatrowski. Durante lo svolgimento del test sul primo gruppo, sono emerse alcune osservazioni che hanno permesso di migliorare l'ambiente di testing: gli studenti hanno consigliato di concatenare il testo derivante dal suggerimento con il contenuto della textarea. Questo ha permesso di migliorare la fase di testing a cui hanno partecipato gli studenti del secondo gruppo e i cui risultati verranno analizzati nella sezione successiva. Per tale motivo, durante la fase di valutazione verranno considerati solo i dati ottenuti durante lo svolgimento sul secondo gruppo.

Per ottenere le informazioni mostrate nella tabella 6.1 è stato fatto compilare un form online prima dell'inizio del test. Il form, anonimo, richiedeva le seguenti informazioni: età, sesso, nazionalità e livello d'inglese. Durante il test, inoltre, è stato contato il tempo impiegato per da ciascun tester per il completamento del test (il valore non include il tempo impiegato per compilare i questionari).

Sesso	**Nazionalità**	**Età**	**Livello Inglese**
$tempo \leq 20min$			
F	Francia	21	B2
M	Finlandia	22	C1
M	Germania	20	C1
F	Russia	24	C1
F	Francia	20	C1
$20min < tempo \leq 30min$			
M	Finlandia	23	B2
F	Francia	19	B2
F	Francia	19	B1
F	Francia	24	C1
M	Belgio	23	C1
F	Olanda	22	C1
M	U.K.	22	C1
$30min > tempo$			
F	Italia	24	B2
M	Finlandia	24	B2
M	Finlandia	25	C2
F	Germania	23	C1

Tabella 6.1: Informazioni degli utenti del secondo gruppo

6.2 Risultati dei test

Nella sezione corrente verranno presentati i risultati ottenuti durante la fase di testing e per la precisione quelli prodotti dalla seconda iterata in quanto, come già detto, l'obiettivo della prima è stato quello di verificare il funzionamento dell'interfaccia e di ottenere opinioni utili su di essa. Ogni utente ha effettuato il test con lo stesso insieme di task ma con ordine differente inoltre l'ordine con cui venivano mostrati i tre sistemi di suggerimento cambiava a seconda dell'utente in modo da evitare che venisse scelta sempre la prima tecnica.

6.2.1 Valutazione dei questionari SUS

Il numero di questionari SUS compilati è pari a 18 la cui distribuzione è la seguente:

- 4 tester, avendo preferito l'inserimento *Manual*, non hanno risposto ad alcun questionario;
- 3 tester non si sono sbilanciati e hanno assegnato a tutte le scelte lo stesso punteggio e questo significa che ognuno dei 3 tester ha compilato 3 questionari, uno per ogni tecnica;
- 4 tester hanno compilato, esclusivamente, il questionario su Minhash;
- 5 tester hanno compilato, esclusivamente, il questionario su MRCTA;

La tabella seguente evidenzia meglio la distribuzione dei questionari effettuati i cui dettagli sono mostrati nei dettagli nelle tabelle della pagina successiva contenente i punteggi assegnati alla domande.

TLSH	**MRCTA**	**Minhash**	**Tot.**
0 + 3	5 + 3	4 + 3	18

TLSH

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Tot.**	**Tot.(x2.5)**
#1	2	3	1	0	2	3	1	0	0	2	14	35
#2	0	0	0	3	0	1	0	0	0	3	7	17,5
#3	0	0	0	0	1	1	0	0	0	2	4	10

Tabella 6.2: Tabella risultati questionario SUS per TLSH

MRCTA

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Tot.**	**Tot.(x2.5)**
#1	2	4	1	2	3	4	3	2	1	2	24	60
#2	1	1	1	3	1	2	1	3	1	3	17	42,5
#3	0	0	2	0	4	1	2	0	0	0	9	22,5
#4	1	0	0	1	0	2	0	0	0	3	7	17,5
#5	3	3	3	4	3	2	4	3	3	3	31	77,5
#6	2	2	3	2	4	1	4	3	3	3	27	67,5
#7	1	1	1	2	2	3	2	2	2	1	17	42,5
#8	2	1	1	0	3	3	4	2	1	1	18	45

Tabella 6.3: Tabella risultati questionario SUS per MRCTA

Minhash

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Tot.**	**Tot.(x2.5)**
#1	2	2	1	0	3	1	3	1	1	0	14	35
#2	0	2	2	0	1	0	0	4	0	2	11	27,5
#3	3	0	2	1	2	0	3	2	3	3	19	47,5
#4	0	0	0	2	0	1	1	0	0	3	7	17,5
#5	1	1	1	3	1	1	0	0	0	2	10	25
#6	2	3	3	4	2	3	3	2	2	4	28	70
#7	2	3	2	4	3	1	3	2	2	4	26	65

Tabella 6.4: Tabella risultati questionario SUS per Minhash

I punteggi SUS finali per ogni tecnica sono mostrati nella tabella seguente:

	SUS avg
TLSH	20.83
Minhash	41.07
MRCTA	46.87

6.2.2 Valutazione delle risposte

In questa sottosezione verranno mostrati i risultati relativi la qualità delle risposte fornite dagli utenti durante lo svolgimento del test. Definiamo task *completato* (1) se la risposta fornita risulta coerente con la descrizione fornita, altrimenti il task verrà marcato come *non completato* (0).

	TASK						
	1	**2**	**3**	**4**	**5**	**6**	
U1	1	1	1	1	1	1	
U2	1	1	0	1	1	1	
U3	1	1	1	1	1	1	
U4	0	0	0	0	0	0	
U5	1	1	1	1	1	1	
U6	1	1	1	1	1	1	
U7	1	1	1	1	1	1	
U8	0	1	1	0	1	1	
U9	0	1	1	1	1	1	
U10	1	1	1	1	1	1	
U11	1	1	1	1	1	1	
U12	1	1	1	1	1	1	
U13	0	0	0	0	0	0	
U14	1	1	1	1	1	1	
U15	1	1	1	1	1	1	
U16	1	1	1	1	1	1	
Totale	12	14	13	13	14	14	**80**

Il numero di task completati (su 96 totali) è di 80 circa l'83%. Come detto nella struttura del test, il tester aveva la possibilità di inserire il testo del suggerimento. Per 36 volte gli utenti hanno optato per l'inserimento manuale indicando, quindi, alcuna preferenza per i suggerimenti; 27 volte sono stati inseriti suggerimenti della tecnica *Minhash* e per 29 volte quelli di *MRCTA*; solo 4 volte gli utenti hanno scelto di inserire i suggerimenti di *TLSH*.

6.2.3 Valutazione punteggi e opinioni

Come già detto, durante la spiegazione della struttura del test, alla fine di ogni task l'utente doveva esprimere la sua esperienza attraverso l'assegnamento di 12 punti da distribuire tra le 4 diverse opzioni (le tre tecniche e l'inserimento manuale).

Nella tabella 6.5 sono mostrati i punteggi totali assegnati ad ogni tecnica alla fine dei sei task. I punteggi vengono utilizzati come criterio di scelta per il questionario SUS da compilare e nella tabella sono evidenziati in grassetto i punteggi con valore maggiore per ogni riga (gli utenti). Il numero di valori in grassetto identifica il numero di questionari SUS compilati e quindi otteniamo, come detto nella sottosezione 6.2.1, 5 questionari per *MRCTA* e 4 per *Minhash*. Le righe che non hanno alcun valore in grassetto identificano gli utenti che hanno compilato tutti e 3 i questionari.

Nella figura 6.4 è mostrato un grafico a torta rappresentante la distribuzione degli score totali forniti per ogni tecnica. Osservando il grafico si può notare che il 31% dei punti totali sono indirizzati verso l'utilizzo inserimento manuale e dietro di esso, con qualche punto di percentuale in meno, sono presenti *MRCTA* e *Minhash* rispettivamente. Per la tecnica *TLSH* sono stati attribuiti solo 160 punti e questo, molto probabilmente, è dovuto alla forte limitazione tecnica dei 512 caratteri necessari richiesti dal metodo.

	MRCTA	Minhash	TLSH	Manual
1	12	**33**	9	18
2	9	**31**	4	28
3	12	**35**	11	14
4	16	**25**	16	15
5	5	6	0	**61**
6	17	14	14	**27**
7	18	18	18	18
8	**30**	25	5	12
9	**34**	22	11	5
10	10	21	14	**27**
11	**42**	10	1	19
12	**30**	12	5	25
13	**22**	19	13	18
14	18	18	18	18
15	27	8	3	**34**
16	18	18	18	18
SOMMA	320	315	160	357

Tabella 6.5: Score totale per ogni task

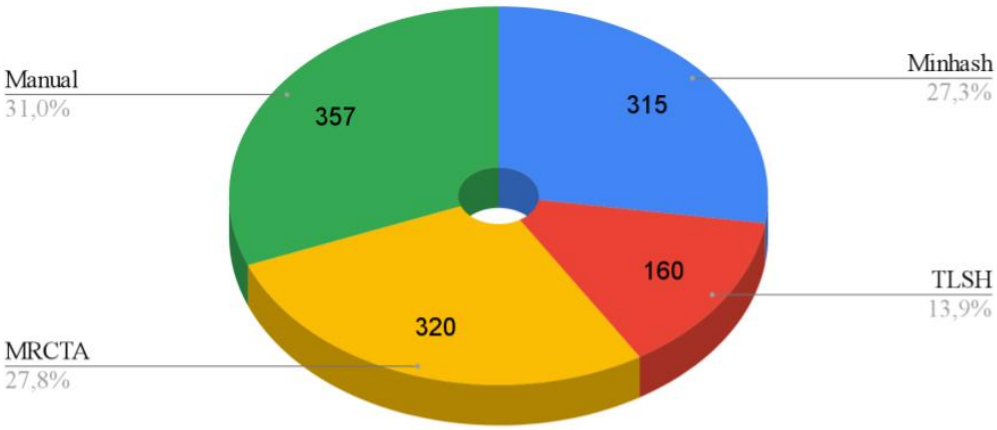


Figura 6.4: Distribuzione dei punteggi forniti

Per quanto riguarda le opinioni finali sono emerse delle osservazioni positive che danno man forte alla tesi della dissertazione. Alcuni utenti hanno trovato molto utile disporre di un sistema che fosse in grado di suggerire loro dei frammenti di testo relativi a ciò che stavano scrivendo e hanno mostrato il loro interesse nel voler utilizzare in futuro il sistema nella sua versione finale:

- *In my opinion it is a great working system, which would only need some user advices in order to be used more easily*;
- *MRCTA has helped me with huge chunks of text multiple times. Suggestions were helpful and a few times the algorithm helped me with the whole text even without much of my own input. I would use this system in its final form.*

Altri, invece, hanno considerato il sistema troppo complesso e hanno trovato utili i suggerimenti solo per scopi informativi senza averli inseriti:

- *It was complicated to use, but this experiment was interessant*;
- *I found it the easiest just to read the statements made in the suggestions as they were often already well written. The automatically generated sentences were of no benefit for me.*

I risultati ottenuti dai questionari SUS e dai punteggi finali sono sintomo che nessuna delle tre tecniche abbia reso l'utente soddisfatto nell'utilizzo del sistema e ciò indica che le tecniche, efficienti in termini di tempo e precisione, non siano riusciti a produrre quei risultati che gli utenti si aspettavano sebbene fossero risultati con un alto valore di similarità. Dalle opinioni finali, invece, è emerso un alto grado di interesse per un sistema di questo genere e questo porta a confermare la tesi, posta all'inizio di tale elaborato. Sebbene ci sia ancora molto lavoro da fare, la creazione di un meccanismo di supporto ha reso la fase di editing dei documenti strutturati, di tema giuridico nel caso di studio in questione, più veloce. Nel capitolo seguente verrà riepilogato il lavoro svolto e gli argomenti trattati nel corso dei vari capitoli con la descrizione di alcuni possibili sviluppi futuri rivolti a migliorare il sistema.

Capitolo 7

Conclusioni e sviluppi futuri

Questo elaborato è iniziato introducendo la seguente tesi: *l'utilizzo di metodologie basate sul calcolo della similarità può migliorare l'editing di documenti strutturati attraverso attività di ricerca, recupero e confronto che forniscono supporto testuale a vari livelli di dettaglio.* Tale affermazione ha influenzato e indirizzato questo lavoro che poneva come obiettivo quello di accettare o confutare la tesi precedente giustificando tale risposta con delle motivazioni solide e ammissibili. Questo obiettivo è stato raggiunto in modo costruttivo, sviluppando un percorso che ha inizialmente fornito i concetti base al lettore per la piena comprensione per poi entrare nei dettagli fino alle descrizione delle implementazioni e metodologie adottate.

Nei capitoli iniziali sono state trattate tematiche e concetti essenziali quali il significato di similarità e la rappresentazione del testo in uno spazio vettoriale. La similarità è stata vista sotto due punti di vista: sintattica e semantica. La prima mira a quantificare quanto simili siano due testi a livello di caratteri mentre la seconda, più difficile, mira a identificare le relazioni, più astratte, tra gli oggetti di riferimento. Come sviluppo futuro, senza dubbio, può essere presa in considerazione la strada della similarità semantica andando ad addestrare reti neurali per fare *Word Embedding*. In letteratura esistono vari modelli come *Word2Vec* [MCC13] e *Doc2Vec* [LM14]. Per questo progetto di tesi è stata presa in considerazione solo la similarità sintattica e per tale fine è stata mostrata, una delle tecniche più popolari in letteratura, chiamata TF-IDF la quale consente di assegnare un punteggio

alle varie componenti (token) presenti nel testo che si desidera vettorializzare. Come abbiamo visto l'utilizzo della tecnica TF-IDF porta con se un grande svantaggio ossia quello legato alla dimensione: se l'insieme dei token della collezione ha cardinalità N allora ogni documento avrà una rappresentazione lineare con tale grandezza. Questo punto non è ammissibile in grandi dataset di documenti in cui l'universo delle parole può avere un valore N elevato.
Per risolvere questo aspetto è stato introdotto e formalizzato il *Nearest Neighbors problem* (capitolo 4) che ci ha permesso di descrivere lo schema *Locality Sensitive Hashing* (sezione 4.3). Questo metodo permette di ridurre la dimensionalità adottando la seguente idea: applicare una funzione di hash agli oggetti della collezione in modo da far collidere, con alta probabilità, elementi simili negli stessi contenitori (bucket). Questo schema permette di ottenere un numero di bucket infinitamente ridotto rispetto all'universo dei possibili input. In base a ciò, è stato analizzato, e poi implementato, *Minhash* (sezione 4.3.1) alle cui fondamenta sono presenti i concetti di LSH.
Minhash è una delle tre tecniche soggette al test e questa permette di ridurre la dimensione del documento preservando però le proprietà di similarità, infatti, come abbiamo visto, *Minhash* è in grado di stimare la distanza di Jaccard tra due documenti utilizzando le firme relative. Questo permette di abbassare la computazione necessaria per il calcolo del valore di similarità.
Un altro sviluppo può essere quello di adottare un ulteriore variante LSH che permetta di stimare il valore di una metrica diversa. Ad esempio, una potenziale tecnica è *Simhash* [Cha02] che consente di calcolare una stima della *cosine distance*.
Oltre ad aver presentato LSH, è stata descritta una tecnica che consente di ridurre il numero di dimensioni del vettore attraverso l'utilizzo del metodo TF-IDF. La tecnica, *Multi-Reference cosine text algorithm (MRCTA)* (sezione 4.4), è composta da due fasi: nella prima si crea un testo di riferimento che verrà utilizzato nella seconda fase che prevede la generazione delle firme. Naturalmente le performance della tecnica MRCTA dipendono fortemente dalla qualità del *reftext* e per l'esperimento sono state considerati gli N 3-words più utilizzati nella collezione di riferimento perciò, in futuro, si potranno analizzare e implementare delle metodologie per la creazione di

un *reftext* che migliori le performance ottenute in questo primo studio.

Nel capitolo dedicato al contributo offerto (capitolo 5) sono state spiegate le due implementazioni delle tecniche ed è stato definito uno standard che permette l'integrazione nell'ambiente, sia di sviluppo che di test, di qualunque altra tecnica. Nel capitolo, inoltre, è stata descritta la fase di normalizzazione della collezione di documenti che, nel caso di studio, aveva come tema l'ambito legislativo. Anche sulla fase di normalizzazione è possibile ipotizzare qualche altro sviluppo futuro come, ad esempio, l'applicazione di una fase di analisi logica sui testi della collezione in tal modo da rilevare il soggetto delle frasi e inserirci un *placeholder* al fine di catturare la struttura della frase o del paragrafo. Un'ulteriore modifica nella fase di normalizzazione potrebbe essere quella di interscambiare l'operazione di *lemming* con quella di *stemming* e vederne gli eventuali benefici.

Infine, nell'ultimo capitolo sono state descritte le fasi di test e valutazione. Dai test effettuati agli studenti di Giurisprudenza, è emerso che un sistema del genere può avere un forte impatto sulla scrittura di documenti suscitando il forte interesse nell'utilizzo di una versione finale del sistema. Sebbene i commenti e le opinioni siano stati positivi, i risultati provenienti dai questionari SUS hanno dimostrato che serva ancora molto lavoro per la creazione di un sistema che possa fornire dei suggerimenti utili all'utente e saranno fondamentali gli esperimenti e gli sviluppi futuri relativi alle due tecniche descritte in questo contributo ossia *Minhash* e *MRCTA*.

In questo lavoro di tesi ho quindi evidenziato, tramite l'ambiente SHE, come l'utilizzo di un supporto testuale possa migliorare la fase di editing di documenti strutturati attraverso attività di ricerca, recupero e confronto che fornisco all'utente suggerimenti correlati al testo che si sta scrivendo. Inoltre, ho analizzato e sviluppato due metodologie basate la prima sul calcolo della distanza di Jaccard (Minhash) e la seconda sulla cosine distance (MRCTA) e ho definito un modello comune per permettere l'integrazione di tecniche all'interno dell'ambiente SHE da me progettato.

Riferimenti

[AIR18] A. Andoni,P. Indyk e I. Razenshteyn. *Approximate Nearest Neighbor Search in High Dimensions*. ArXiv 1806.09823, 2018.

[AMN+98] S. Arya, D. Mount, N. Netanyahu, R. Silverman, e A. Wu. *An optimal algorithm for approximate nearest neighbor searching fixed dimensions.* Journal of the ACM (JACM). 1998.

[BCG05] M. Bawa, T. Condie e P. Ganesan. *LSH Forest: Self-Tuning Indexes for Similarity Search.*International World Wide Web Conference Committee (IW3C2). 2005.

[Ben75] J. Bentley. *Multidimensional binary search trees used for associative searching*. Commun. ACM 18. 1975.

[Bro97] A. Z. Broder. *On the resembance and containment of documents*. In *Proceedings of the Compression and Complexity of Sequences 1997*. 1997.

[Cha02] M. Charikar. *Similarity estimation tecniques from rounding algorithms*. In *STOC*. 2002.

[GC19] G. Giusti. *Similarità tra stringhe, applicazione dell'algoritmo TLSH a testi di ingegneria*. 2019.

[GF13] W. Gomaa e A. Fahmy. *A Survey of Text Similarity Approaches*. International Journal of Computer Applications. 2013.

[IM98] P. Indyk e R. Motwani. *Approximate nearest neighbors: Towards removing the curse of dimensionality*. 1998.

[Jon72] K. Jones. *A statistical interpretation of term specificity and its application in retrieval*. Journal of documentation. 1972.

[Kle97] J. Kleinberg. *Two algorithms for nearest-neighbor search in high dimensions.* In Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. 1997.

[LM14] Q. Le e T. Mikolov. *Distributed representations of sentences and documents*. In *Proceedings of the 31st International Conference on International Conference on Machine Learning*. 2014.

[LS09] J. Lewis e J. Sauro. *The Factor Structure of the System Usability Scale*. Proceedings of the 1st International Conference on Human Centered Design: Held as Part of HCI International. 2009.

[MCC13] T. Mikolov, K. Chen, G. S. Corrado e J. Dean. *Efficient Estimation of Word Representations in Vector Space*. CoRR abs/1301.3781, 2013.

[MN18] H. Mohammadi e Amin Nikoukaran. *Multi-reference Cosine: A New Approach to Text Similarity Measurement in Large collections*. ArXiv:1810.03099, 2018.

[MK19] H. Mohammadi e S.H. Khasteh. *A Fast Text Similarity Measure for Large Document Collections using Multi-reference Cosine and Genetic Algorithm*. ArXiv:1810.03102, 2019.

[OBB18] T. Ojeda, R. Bilbro, B. Bengfort. *Applied Text Analysis with Python*. O'Reilly Media, Inc. 2018.

[SLY18] O. Shahmirzadi, A. Lugowski e K. Younge. *Text Similarity in Vector Space Models: A Comparative Study*.CoRR,abs/1810.00664. 2018.

[SM08] M. Slaney e M. Casey. *Locality-Sensitive Hashing for Finding Nearest Neighbors*. Lecture Notes. *Signal Processing Magazine*. 2008.

[VK16] M. Vijaymeena e K. Kavitha. *A Survey on Similarity Measures in Text Mining*. Machine Learning and Applications: An International Journal. 2016.

[Zip49] G. Zipf.*Human behavior and the principle of least effort*. Addison-Wesley Press. 1949.