Finding Similar Itmes - Document Similarity

Consideriamo un primo problema di data mining: la ricerca di elementi simili in un insieme di dati di grandi dimensioni. Supponiamo di avere in input un insieme $X_1, X_2, \ldots, X_N$, dove N è dell'ordine di milioni o miliardi. Ogni elemento è rappresentato da dati multidimensionali; ad esempio, un'immagine può essere rappresentata come una matrice di pixel, che a sua volta può essere trasformata in un vettore appartenente a uno spazio di dimensione h. Anche h è dell'ordine di milioni, il che significa che i dati in input non solo sono numerosi, ma anche ad alta dimensionalità.

Inanzitutto bisogna introdurre una funzione di distanza, $d(X_1, X_2)$ che quantifica la "distanza" tra due elementi X_1 e X_2. L'obiettivo è **trovare tutte le coppie** (X_i, X_j) tale che $d(X_i, X_j) \leq S$, dove S è una certa soglia.

Una soluzione banale impiegherebbe tempo $O(N^2 \cdot h)$, che per quanto abbiamo visto prima, con N e h dell'ordine dei milioni, questa soluzione è inammissibile, in quanto troppo inefficiente. Vedremo in seguito un'algoritmo che **magicamente** impiega tempo $O(N \cdot h')$, con $h' << h$.

Document Similarity

Passiamo ora all'argomento **Similar Items**, concentrandoci su un caso specifico per approfondire: il problema **Document Similarity**. Questo implica individuare tutte le coppie di documenti di testo che risultano simili tra loro. Tale problema ha applicazioni molto importanti in vari campi, come:

- Rilevamento di Plagio
- Identificazione di pagine web simili

Scegliere la funzione di distanza

Ogni applicazione necessità di una funzione di distanza approprita, nel caso del document similarity, la funzione di distanza sceltà e la **Jaccard Similarity**.

> ✓ **Success**
>
> - Definiamo la **Jaccard Similarity** di due insiemi come:

$$J.sim(C_1, C_2) = \frac{\left| C_1 \cap C_2 \right|}{\left| C_1 \cup C_2 \right|}$$

- Definiamo la **Jaccard Distance** di due insiemi come:

$$J.d(C_1, C_2) = 1 - J.sim(C_1, C_2)$$

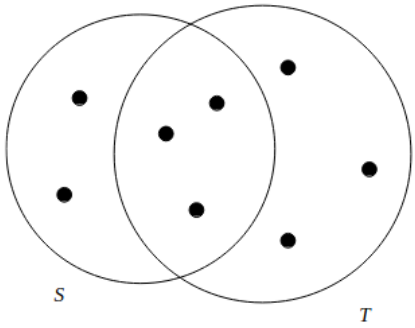


Figure 3.1: Two sets with Jaccard similarity 3/8

La domanda che sorge spontanea è: come possiamo utilizzare la Jaccard similarity, dato che richiede di lavorare su insiemi, mentre noi stiamo trattando documenti di testo?

Ricapitolando, la nostra situazione è la seguente:

Abbiamo $Doc \in Sigma^*$, dove Σ rappresenta l'alfabeto, e dobbiamo trovare un modo per trasformare i documenti in insiemi, così da poter applicare la Jaccard similarity.

Prima di andare nel dettaglio, osserviamo ad alto livello l'algoritmo.

1. **Input**: In insieme grande di documenti;
2. **Shingling**: Convertire i documenti in **insiemi** (molto grandi);
3. **Min-Hashing**: Convertire gli insiemi molto grandi in piccole **firme (signatures)**, mantenendo la Jaccard similarity;
4. **Localitive-Sensitive Hashing (LSH)**: LSH si concentra su coppie di firme che probabilmente appartengono a documenti simili secondo la Jaccard similarity;
5. **Output**: Ritorna le coppie di documenti candidate ad essere simili.

Step 1. Shingling: Convertire i documenti in insiemi

Consideriamo un documento come una stringa di caratteri. Sia Σ l'alfabeto, per esempio l'alfabeto inglese composto da 27 caratteri ($\Sigma = \{a,\ b,\ c,\ \ldots\}$) e sia Σ^* l'insieme di tutte le stringhe costituie con i caratteri di Σ. Sia $Doc \in \Sigma^*$ un documento; definiamo un k-shingle di un documento come una qualsiasi sottostringa (*token, che può essere formato da caratteri o parole*) di lunghezza k all'interno

di quel documento. Allora, associamo a ciascun documento l'insieme composto da k-shingles che appaiono una o più volte nel documento Doc.

Example

Supponiamo che il nostro documento D sia la stringa `abcdabd` e scegliamo $k = 2$. Allora l'insieme 2-shingles per D è $\{ab, bc, cd, da, bd\}$. Osserviamo che `ab` compare 2 volte nel documento, ma una volta sola nell'insieme. Una variazione sarebbe quella di usare multinsiemi invece che insiemi per rappresentare gli shingle.

Ci sono varie possibilità riguardo i caratteri "vuoto", "tabulazione", "nuova riga", un'idea sarebbe quella di sostuire tali caratteri con con semplice carattere vuoto.

Ora dobbiamo scegliere il valore di k. Se scegliamo un k molto piccolo, ad esempio 1 o 2, è probabile che documenti completamente diversi risultino simili tra loro secondo la Jaccard Similarity, portando a molti falsi positivi. Per evitare ciò, la scelta di k non dovrebbe essere casuale.

La regola generale è la seguente:

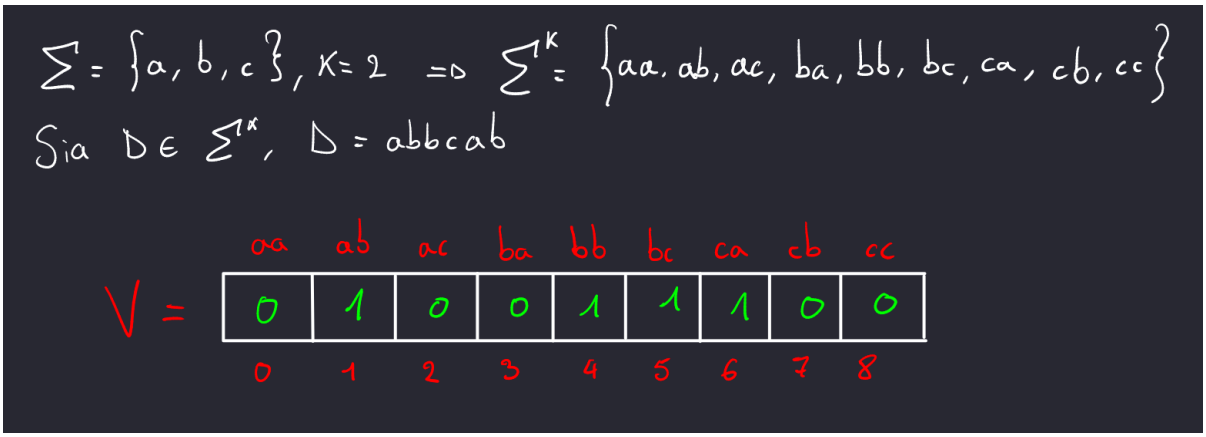
- Il valore di k dovrebbe essere abbastanza grande da ridurre la probabilità che uno shingle specifico compaia casualmente in un documento qualsiasi.

Scegliendo un k maggiore, riduciamo la probabilità che uno shingle si presenti in modo casuale in documenti diversi, rendendo il confronto più preciso. Un valore di k più lungo permette di catturare pattern più specifici, migliorando l'affidabilità della similarità tra documenti.

In caso di documenti piccoli, come email, $k = 5$ è ragionevole. Mentre per documenti più grandi, $k = 9$ è un ottima scelta.

Sia $D \in \Sigma^*$ un documento rappresentato dal suo insieme di k-shingles. Se consideriamo Σ^k l'insieme di tutti i possibili k-shingle ordinato, allora possiamo rappresentare D come un vettore binario di dimensione $\left| \Sigma^k \right|$, dove ogni shingle è una componente di questo vettore e:

$$v[i] = \begin{cases} 1 & \text{se l' i-esimo shingle dell'insieme ordinato } U \text{ sta nel documento} \\ 0 & \text{se l' i-esimo shingle dell'insieme ordinato } U \text{ non sta nel documento} \end{cases}$$



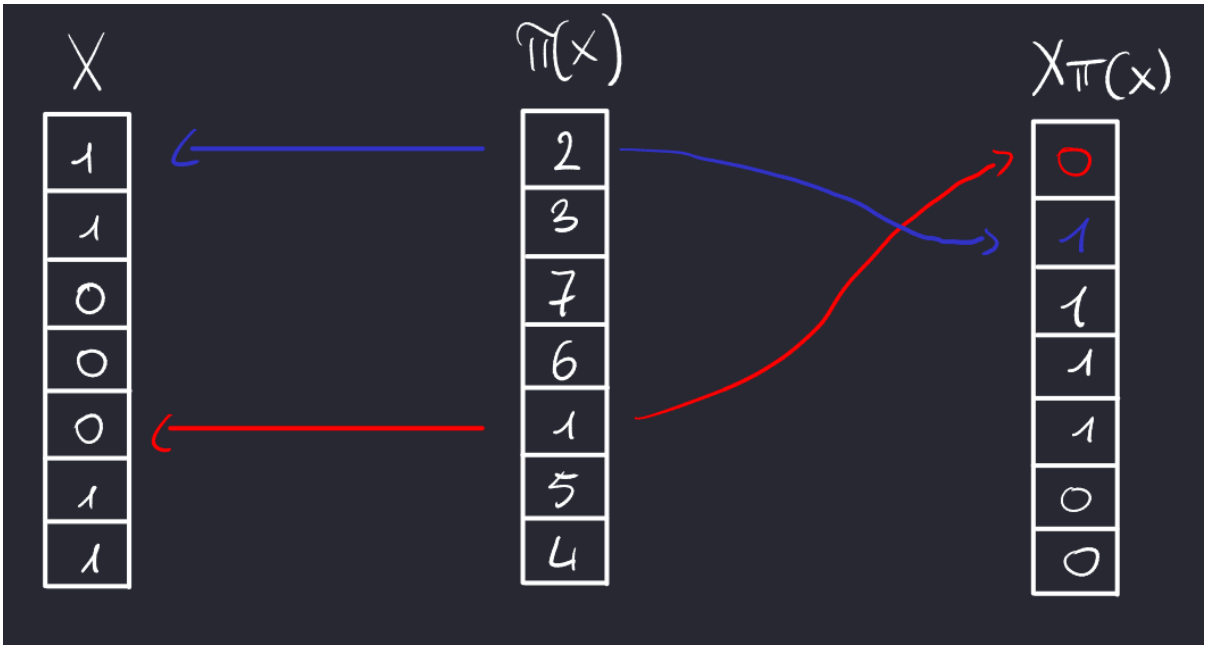
Possiamo rappresentare un insieme di documenti come una collezione di insiemi, dove ciascun insieme è descritto da un vettore binario. Questi insiemi formano la cosiddetta matrice caratteristica, in cui le colonne corrispondono ai documenti (vettori binari) e le righe agli shingle ordinati.

Element	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	1	1	0

Step 2. MinHashing: Convertire gli insiemi molto grandi in piccole firme (signatures), mantenendo la Jaccard similarity

Una volta calcolata la matrice caratteristica, possiamo già iniziare a determinare le coppie di elementi simili. Tuttavia, questo calcolo richiederebbe un tempo elevato, pari a $\theta(N^2 \cdot \left| \Sigma^k \right|)$. Per ridurre questa complessità, possiamo cercare di ridimensionare la matrice, riducendo il valore di $h = \left| \Sigma^k \right|$.

La tecnica utilizzata per questa operazione è chiamata **MinHashing**. Per eseguire il *minhash* di un insieme rappresentato da una colonna della matrice caratteristica, si applica una permutazione casuale delle righe. Il minhash di una colonna è quindi l'indice della prima riga, nella colonna permutata, in cui compare un 1.



Note

Dunque la permutazione $\pi(x)$ si legge in ordine crescente, non dalla prima posizione, ma dall'elemento 1.

Success

Supponiamo che le righe della matrice M sono permutate, secondo una permutazione casuale π.
Def: Per ogni colonna C, definiamo **Min-Hash function** $h_\pi(c)$ l'indice della prima riga (nella permutazione π) in cui nella colonna C appare il valore 1 (è lo chiamiamo $min(\pi(C))$).

Tra minhash e Jaccard similarity c'è una relazione, ovvero:

- La probabilità che la funzione di minhash, applicata su una permutazione casuale delle righe, produca lo stesso valore per due insiemi è uguale alla Jaccard similarity tra questi insiemi.

Questo significa che, se abbiamo due insiemi e applichiamo una permutazione casuale delle righe della matrice caratteristica, la probabilità che il minhash (ovvero l'indice della prima riga con un 1) sia lo stesso per entrambi gli insiemi è esattamente uguale al valore della similarità di Jaccard tra di loro.

- Se $J.sim(C_1, C_2)$ è **alta**, allora con alta probabilità $h(C_1) = h(C_2)$
- Se $J.sim(C_1, C_2)$ è **bassa**, allora con alta probabilità $h(C_1) \neq h(C_2)$

Success

Teorema - J-Similarity Preserving: Fissate C_1 e C_2 due colonne della matrice caratteristica. Scelta uniformemente random una permutazione π. Allora

$$Pr_\pi[h_\pi(C_1) = h_\pi(C_2)] = J.sim(C_1, C_2)$$

Consideriamo un documento X, cioè una colonna nella matrice caratteristica. Ogni riga di X corrisponde a uno *shingle* (per esempio, una sottostringa di testo), e la cella è impostata a 1 se lo shingle appartiene al documento X, 0 altrimenti. Pertanto, possiamo vedere X come un insieme di shingle, con $y \in X$ che indica uno shingle presente in X (ossia, la cella in corrispondenza di y è posta a 1).

Dimostrazione:

Passo 1: Calcolo della probabilità che uno shingle sia il minimo in una permutazione.

Per una permutazione casuale π, la probabilità che uno shingle specifico $y \in X$ venga mappato come *minhash* della colonna X è data da:

$$\Pr[\pi(y) = \min(\pi(X))] = \frac{1}{|X|}$$

poiché ogni shingle $y \in X$ ha uguale probabilità di essere il minimo rispetto a π.

Passo 2: Calcolo della probabilità che $h_\pi(C_1) = h_\pi(C_2)$.

Per stimare $J.sim(C_1, C_2)$, dobbiamo calcolare la probabilità che il *minhash* delle due colonne C_1 e C_2 sia lo stesso, ovvero $\Pr_\pi[h_\pi(C_1) = h_\pi(C_2)]$.

1. Sia y l'elemento tale che $\pi(y) = \min(\pi(C_1 \cup C_2))$. Ciò significa che y è l'elemento con il valore minimo nella permutazione unione $C_1 \cup C_2$.
2. Notiamo che:

- Se $y \in C_1$, allora $\pi(y) = \min(\pi(C_1))$.
- Se $y \in C_2$, allora $\pi(y) = \min(\pi(C_2))$.

In altre parole, il valore minimo della permutazione per C_1 o C_2 sarà dato da uno shingle presente in almeno una delle due colonne.

3. Il caso favorevole, in cui $h_\pi(C_1) = h_\pi(C_2)$, si verifica se e solo se $y \in C_1 \cap C_2$ (ossia, y è uno shingle presente in entrambe le colonne C_1 e C_2). In tal caso, y sarà il minimo in

entrambe le colonne.

4. Poiché la permutazione π è uniforme e casuale, possiamo dire che la probabilità che il minimo di $\pi(C_1 \cup C_2)$ appartenga sia a C_1 sia a C_2 è proprio la probabilità di intersezione rispetto all'unione degli shingle, cioè:

$$\Pr_{\pi}[h_\pi(C_1) = h_\pi(C_2)] = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|} = J.sim(C_1, C_2)$$

Conclusione

Abbiamo dimostrato che la probabilità che il *minhash* di C_1 sia uguale al *minhash* di C_2 per una permutazione casuale π è esattamente la Jaccard Similarity tra C_1 e C_2. Questo conclude la dimostrazione.

Dalla matrice caratteristica iniziale possiamo ottenere una matrice molto più piccola, chiamata **matrice delle firme (signature matrix)**. Se applichiamo una sola permutazione casuale, la matrice delle firme sarà costituita da N colonne e una sola riga. Tuttavia, aumentando il numero di permutazioni e quindi il numero di firme per ciascuna colonna, possiamo migliorare la stima della Jaccard Similarity. Questo processo di miglioramento della "fiducia" nella stima di similarità si basa sul concetto di *concentrazione intorno al valore atteso*. Di conseguenza, per ciascuna colonna otteniamo una serie di Min-Hash signatures.

Definiamo quindi la firma di un documento C come un vettore di più Min-Hash signatures indipendenti:

$$SIG(C) = \langle h_{\pi_1(C)}, h_{\pi_2(C)}, \ldots, h_{\pi_t(C)} \rangle$$

dove t è un numero molto grande di permutazioni $(t >> 1)$. Questa raccolta di firme consente di aumentare la precisione nella stima di similarità tra due colonne.

Per due vettori di signature, $SIG(C_1)$ e $SIG(C_2)$, definiamo la similarità $Sign - Sim$ come la frazione di signature scelte in cui C_1 e C_2 sono d'accordo (cioè hanno lo stesso Min-Hash signature). Dunque, si calcola la *media* delle similarità ottenute su tutte le permutazioni considerate. Al crescere di *t*, questa media tende al valore atteso, ovvero alla Jaccard Similarity.

Random-Algorithm-Doc-Pair-Check(Col_1, Col_2, T)

- Col_1, Col_2 colonne della matrice caratteristica
- T parametreo di confidenza

1. Scegli una permutazione uniformemente random $\pi_j \in \Pi_m$

2. Calcola il *minhash* $h_{\pi j}(C_1)$ e $h_{\pi j}(C_2)$
3. Return $Sign - Sim(C_1, C_2) = \frac{\left|\{\, j : h_{\pi j}(C_1) = h_{\pi j}(C_2) \,\}\right|}{t}$

> ✓ **Success**
>
> **Corollario**: Per t che tende a ∞, $Sign - Sim(C_1, C_2) = J.sim(C_1, C_2)$.

Complessità spaziale MinHash

Scegliamo un $t = 100$ fisso di permutazioni random delle righe. Questo numero di permutazioni viene utilizzato per generare una firma (o sketch) che rappresenta il documento o l'insieme in maniera compatta.

Sia $SIG(i, C)$ l'indice della prima riga che contiene un valore 1 nella colonna C secondo la permutazione casuale i. Ogni valore $SIG(i, C)$ occupa spazio proporzionale a $\theta(log(|C|)) = \theta(log(m))$ dove m rappresenta il numero totale di possibilit $k - shingles$. L'insieme di tutti questi $SIG(i, C)$, cioè $SIG(*, C)$ rappresenta una firma del documento o dell'insieme C. Questa firma è compatta, circa 100 byte. In termini generali, la dimensione della firma è proporzionale a $theta(t\ log(m))$.

In coclusione, grazie al minhash si è riusciti a "comprimere" i vettori di bit lunghi in firme più corte, cioè in **sketch**, in pratica si passa da un vettore di dimensione $m = |\Sigma|^k$ ad un vettore $t\ log(m)$.

Ottimizzazione MinHash

La fase di Min-Hash, quindi di comprimere la matrice grande richiede di creare diverse permutazioni random delle righe della matrice originale per ciascun documento. Sapendo che $m = |\Sigma|^k$, e supponendo di usare l'alfabeto inglese e $k = 10$, allora anche generare una **SOLA** permutazione per 27^{10} righe sarebbe troppo dispendioso in termini di tempo e risorse.

Invece di generare t permutazioni, si sostituiscono le permutazione con le funzioni hash randomizzate (**Randomized Hash Functions**). Si scelgono t funzioni hash f_i. Ogni funzione hash $f_i : [m] \to [m]$, mappando le righe in modo casuale (ci potrebbe essere delle collisioni, ma vedremo che la probabilità delle collisioni è molto bassa).

Algortimo per il calcolo del MinHash

```
// Input: Original matrix M of size m x N, number of hash functions t
// Output: Signature matrix SIG of size t x N

// Step 1: Initialize the signature matrix
for each column C = 1 to N do
    for each hash function i = 1 to t do
        SIG[i][C] = ∞    // Initialize SIG(i, C) to infinity

// Step 2: Populate the signature matrix
for each row j = 1 to m do
    for each column C = 1 to N do
        if M[j][C] == 1 then (**)           // If the element in M is 1
            for each hash function i = 1 to t do
                h_value = f_i(j)        // Compute the hash function i on row j
                if h_value < SIG[i][C] then
                    SIG[i][C] = h_value // Update SIG(i, C) with the minimum hash value
```

Osserviamo inanzitutto che questo algoritmo impiega tempo $\theta(mN)$. L'ottimizzazione di questo algoritma sta nel fatto che non generiamo pià t permutazioni random delle m righe, ma generiamo invece t funzioni hash. Infatti per generare una permutazione delle m righe impieghiamo tempo $O(nlog(n))$ mentre per generare una singola funzione hash impieghiamo tempo $log(m)$. Infatti:

$$\mathcal{H} = \{h_{a,b} : [m] \to [m]\}$$

dove lo spazio per memorizzare una singola hash function è $O(log(m))$ e il tempo per generarla è $O(log(m))$.

$$\Pi(log(m!)) \approx mlog(m)$$

dove lo spazio per memorizzare una singola permutazione è $O(m)$ e il tempo per generarla è $O(mlog(m))$.

Inoltre, ponendo la condizione (**), possiamo usare la proprietà che la matrice caratteristica è molto sparsa, andando a calcolare la funzione hash solo per quelle righe in cui c'è un 1.

Step 3. Locality-Sensitive Hashing: Si concentra su coppie di firme che probabilmente appartengono a documenti simili

Anche se possiamo utilizzare il minhashing per comprimere documenti di grandi dimensioni in firme ridotte e preservare la similarità attesa tra qualsiasi coppia di documenti, potrebbe comunque essere impossibile trovare in modo efficiente le coppie con la maggiore similarità. Il motivo è che il numero di coppie di documenti potrebbe essere troppo elevato, anche se i documenti non sono particolarmente numerosi.

> **Example**
>
> Supponiamo di avere un milione di documenti e di utilizzare firme di lunghezza 250. In questo caso, utilizziamo 1000 byte per documento per le firme, e l'intero dataset occupa un gigabyte – meno della memoria principale tipica di un laptop. Tuttavia, ci sono $\binom{1.000.000}{2}$, ovvero mezzo trilione di coppie di documenti. Se occorre un microsecondo per calcolare la similarità tra due firme, sarebbero necessari quasi sei giorni per calcolare tutte le similarità su quel laptop.

Dunque l'obiettivo principale dell'approccio **LSH** è ridurre il numero di confronti necessari per identificare coppie simili in un insieme di elementi, evitando il confronto tra ogni coppia possibile.

L'idea generale è di usare una funzione hash universale usata per "hashare" gli elementi in una hash table in modo tale che gli elementi simili abbiano maggiore possibilità di essere hashati nello stesso bucket rispetto agli elementi diverse.

Alla fine di questo procedimento, per vedere quali sono le possibili coppie *candidate*, andiamo a considerare ciasun bucket, sapendo che in un bucket sono stati mappati tutte quelle possibili coppie candidate ad essere simili. L'obiettivo è che la maggior parte delle coppie diverse non venga mai hashata nello stesso bucket e quindi non venga mai controllata. Le coppie diverse che invece vengono hashate nello stesso bucket sono falsi positivi. Tali coppie possono essere poi facilmente controllate alla fine, andando a fare una ricerca nella matrice caratteristica, ovvero nella matrice degli shingling. Allo stesso modo, si spera che la maggior parte delle coppie effettivamente simili venga hashata nello stesso bucket. Quelle che non lo fanno sono falsi negativi, e l'obiettivo è di minimazzare il numero di tali coppie, poiché è molto difficile andare a controllare quali siano.

- **Falso Positivo**: Sono tutte quelle coppie che non risultano simili ma che vengono mappate nello stesso bucket. Qui basta semplicemente prendere tutte le coppie candidate che sono molto di meno rispetto N documenti inziali, e andare ad efettuare un ulteriore controllo nella matrice caratteristica.

- **Falso Negativo**: Sono tutte quelle coppie che risultano simili ma che vengono mappate in bucket diversi. Qui il problema sta nel identificare quali siano queste coppie.

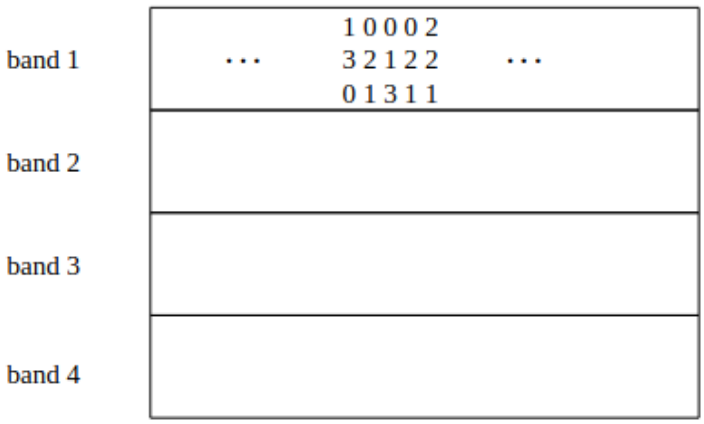
Quindi, dato in input una soglia limite s $(0 < s < 1)$, allora

> **Info**
>
> **Def**: Due colonne x e y sono dette **coppie candidate** se la loro firma $SIG(*, x)$ e $SIG(*, y)$ concordano su almeno una frazione s delle loro righe, ovvero:
>
> $$\frac{|\{i \in [t] : SIG(i, x) = SIG(i, y)\}|}{t} \geq s$$

> **Note**
>
> La similarità tra le firme di due colonne non deriva da piccole differenze in alcuni o diversi valori delle loro righe: ma richiede che un numero significativo di valori (o la maggior parte) delle righe sia identico tra due colonne!

Poiché hashare l'intera colonna delle firme in un unico bucket può generare molti falsi positivi e falsi negativi, possiamo adottare un approccio alternativo basato sempre su LSH.

Quando disponiamo delle firme MinHash per gli elementi, un metodo efficace consiste nel dividere la matrice delle firme in b **bande**, ciascuna formata da r **righe**. Per ogni banda, si calcola un hash unico del vettore di r righe (ovvero la porzione corrispondente della colonna).



Questa divisione introduce un vincolo più forte per considerare due colonne simili: devono avere valori identici in **tutte** le righe di almeno una banda per essere mappate nello stesso bucket. In altre parole:

- Colonne simili (con valori identici nella maggior parte delle righe) hanno una buona probabilità di finire nello stesso bucket in almeno una banda, diventando coppie candidate.
- Colonne diverse difficilmente avranno valori identici in tutte le righe di una banda e quindi saranno escluse come candidate.

Quindi, due colonne sono considerate coppie candidate secondo la Jaccard similarity se almeno una banda è mappata nello stesso bucket. Questo approccio riduce i falsi positivi e negativi rispetto all'hash dell'intera colonna, bilanciando precisione ed efficienza.