SIMILARITA' TRA DOCUMENTI

Algoritmo LSH e MinHash

Federica Bianchi (1891952)

Hazem Dewidar (1883881)

Facoltà ingegneria dell'informazione informatica e statistica
Corso triennale d'Informatica

INTRODUZIONE

Uno dei problemi fondamentali del data-mining è stato da sempre quello di ricercare ed esaminare documenti “simili”. Tale attività, infatti, vede molte applicazioni: un semplice esempio potrebbe essere quello della ricerca di pagine web plagiate oppure l’impiego nel e-commerce. Ad esempio Amazon ha milioni di clienti e vende milioni di articoli. Il suo database tiene traccia di quali articoli ha comprato quale utente. Si può dire che due clienti sono simili se l’insieme rappresentato dai loro acquisti ha un alto coefficiente di Jaccard.

La nozione di similarità a cui abbiamo fatto riferimento non è quella di “significato simile”, ma piuttosto quella che si basa sulla concezione dei documenti come insiemi: due documenti sono tanto più simili quanto è maggiore la dimensione della loro intersezione. Questa nozione di similarità è nota come “Jaccard similarity”. Per trasformare i documenti in insiemi abbiamo utilizzato la tecnica dello “shingling” e abbiamo implementato un algoritmo di minhashing per comprimere gli insiemi associati a ciascun documento, mantenendo però la loro struttura in modo tale da poterne dedurre la similarità anche in una versione più compressa.

LA SIMILARITA’ DI JACCARD PER GLI INSIEMI

Siano A e B due insiemi ,allora la similarità di Jaccard è data dalla formula:

$$\frac{|S \cap T|}{|S \cup T|}$$

ovvero il rapporto fra la cardinalità della loro intersezione e la cardinalità della loro unione. Il valore del coefficiente è 0 quando i due insiemi sono disgiunti, 1 quando sono uguali e compreso tra 0 e 1 altrimenti. **Dunque due insiemi sono più simili quando il loro coefficiente di Jaccard è vicino ad 1**.

Possiamo denotare il coefficiente di Jaccard di A e B come SIM(A, B).

L'IDEA

Descriviamo i passaggi logici che abbiamo seguito per lo sviluppo del nostro programma.

1. Costruiamo gli insiemi di k-shingles per ogni documento

Uno dei metodi più usati per rappresentare i documenti, con lo scopo di individuare la similarità, è quello di costruire gli insiemi di stringhe che appaiono nei documenti stessi. In questo modo, a seconda della dimensione che utilizziamo per creare le stringhe, se dei documenti condividono parole o frasi, avranno più elementi in comune nei loro insiemi, anche se tali stringhe appaiono in ordine diverso nei corrispettivi documenti .

Per ottenere questi insiemi utilizziamo il concetto di shingle: ogni documento è una stringa di caratteri, dunque possiamo definire un "**k-shingle**" come una sottostringa del documento di lunghezza k. Ad esempio se il documento è la stringa "ciao" e scegliamo come dimensione degli shingle 2, otterremo l'insieme:

{"ci", "ia", "ao"}

dove ogni elemento è un 2-shingle del documento.

SCEGLIERE LA DIMENSIONE DEGLI SHINGLES

Possiamo scegliere come dimensione degli shingles una costante qualsiasi; ciononostante se scegliamo k troppo piccola, possiamo aspettarci che molte sequenze di k caratteri appaiano nei documenti, anche se questi in realtà non hanno molte parole o frasi in comune. Dunque la scelta di k dipende, generalmente, dalla lunghezza tipica di alcuni documenti; in generale la cosa importante da ricordare è:

- k deve essere scelto abbastanza grande da far si che la probabilità che qualunque shingle appaia in ogni dato documento è bassa.

2. Hash degli shingles

Invece di utilizzare direttamente le sottostringhe ottenute come shingles, possiamo utilizzare una funzione di hash per mappare le stringhe di lunghezza k in unsigned int. In questo modo i nostri shingles saranno gli hash ottenuti: l'insieme che rappresenta il documento sarà l'insieme degli interi ottenuti passando gli shingles del documento alla funzione di hash. Per effettuare l'hash delle stringhe abbiamo utilizzato la funzione di hash "jenkins one at a time".

3. Minhashing degli shingles

Gli insiemi di shingles sono molto grandi, anche se vengono hashati. Per tale ragione, vorremmo un modo per sostituire questi insiemi di hash con rappresentazioni più piccole chiamate "**signatures**"; il processo che porta alla creazione della matrice delle signature è proprio l'algoritmo di **MinHash.**

Possiamo individuare un'importante connessione tra il minhashing e la similarità di Jaccard degli insiemi su cui effettuiamo il MinHash:

- La probabilità che la funzione di minhash per una permutazione random delle righe della matrice caratteristica produca lo stesso valore per due insiemi, è uguale alla similarità di Jaccard per quei due insiemi.

Per effettuare il minhash di un insieme (rappresentato sotto forma di matrice caratteristica) possiamo prendere una permutazione delle righe: il valore del minhash per ogni colonna è il numero della prima riga, dopo la permutazione, in cui la colonna vale 1.

Ovviamente permutare una matrice caratteristica di grandi dimensioni impiegherebbe molto tempo; fortunatamente, però, è possibile simulare una permutazione randomica delle righe della matrice, mappando ogni riga tramite una funzione di hash. Più funzioni di hash vengono utilizzate, più è probabile che l'hash di documenti simili collida per almeno uno dei componenti. Non c'è la garanzia che questo accada sempre, ma la probabilità che si verifichi è la stessa che si ottiene dalla similarità di Jaccard.

Per costruire la matrice delle signatures, prendiamo ogni documento nella sua rappresentazione come insieme di hashed shingle e facciamo l'hash di ogni hashed

shingle con una funzione di hash. Una volta fatto ciò, troviamo il minimo valore hash prodotto: quello sarà il valore corrispondente per il documento in quella funzione di hash. Questo procedimento viene eseguito per ogni documento e con ogni funzione di hash.

4. Locality-Sensitive Hashing (LSH)

Anche se con il minhashing comprimiamo documenti molto grandi in signature più piccole, rimane comunque molto difficile cercare le coppie di documenti che sono molto simili in maniera efficiente. Questo perché il numero di coppie di documenti da confrontare potrebbe essere molto grande, anche se la loro rappresentazione è stata ridotta.

Un approccio che possiamo utilizzare è quello dell'LSH: effettuiamo l'hash degli hashed shingle della matrice delle signature in dei buckets, in modo tale che documenti abbiano più probabilità di essere hashati nello stesso bucket. I documenti hashati nello stesso bucket vengono considerati come una coppia di documenti candidati per essere simili. In questo modo, non dovremo confrontare ogni coppia di documenti in input, ma solamente i documenti che appartengono allo stesso bucket.

La speranza è che i documenti più differenti non vengano hashati nello stesso bucket e quindi non saranno mai confrontati. I documenti differenti che sono hashati nello stesso bucket sono chiamati falsi positivi; i falsi negativi sono invece documenti simili che NON vengono hashati nello stesso bucket.

Generalmente non vengono hashati singoli elementi della matrice delle signature nei buckets, ma intere bande della matrice. Il numero e la dimensione di queste bande può variare a seconda dei parametri che si vogliono trovare.

IDEA IMPLEMENTATIVA

1. Prendiamo in input la cartella contenente i documenti, la lunghezza degli shingle, il numero di permutazioni da effettuare (cioè il numero di minHash) che deve essere sottoposto ogni documento, il numero di righe per banda, il numero di buckets in cui hashare nel LSH ed, eventualmente, il numero di thread.
2. Convertiamo i file in lettura memorizzandoli in array di stringhe.
3. Creiamo i shingles
4. Creiamo la signature matrix
    a. Per ogni documento, hashiamo ogni shingle con la funzione di jenkins.
    b. Per l'i-esimo hash e per il j-esimo documento, diamo in input a una funzione che crea la signature matrix i risultati ottenuti precedentemente dalla jenkins e ci salviamo nella posizione (H_i, D_j) della signature matrix il minimo (tra gli hash).
5. Applichiamo l'LSH per cercare di diminuire il numero di documenti confrontati allo stretto necessario.
    a. Dividiamo la signature matrix in r righe (di conseguenza avremo un numero di bande b = numero di Hashes/r).
    b. Per ogni banda, hashiamo i documenti in dei buckets. Nel caso due o più documenti finissero nello stesso bucket, questi verranno confrontati calcolando il coefficiente di Jaccard.

    c. Per confrontare i documenti che sono stati hashati nello stesso bucket, usiamo gli hashed shingles ottenuti precedentemente dalla jenkins. Eliminiamo prima i duplicati da entrambi i documenti, poi effettuiamo l'intersezione ed infine applichiamo il principio di inclusione-esclusione per trovarci l'unione ($|A| + |B| - |A \cap B|$).

    d. Ci salviamo i risultati nella jaccard matrix

SCELTE IMPLEMENTATIVE PTHREAD

Abbiamo deciso di dividere il lavoro tra i thread basandoci sul loro rank. Ogni thread lavora su un subset del problema: dapprima si occupa della gestione di un sottoinsieme di documenti (come ad esempio l'allocamento degli alcune righe delle matrici, la conversione da file in array di char, la creazione dei shingle e della signature matrix). Dopo che ogni thread ha finito la gestione di ogni documento, aspetterà (attraverso una barriera) tutti gli altri thread; infatti poichè ogni thread calcola una parte della matrice delle signature e l'LSH utilizza tale matrice, prima di poter invocare l'LSH è necessario che tutti i thread abbiano concluso il loro lavoro. Infine chiameranno la funzione LSH. Abbiamo usato un mutex per proteggere la matrice caratteristica dei bucket ogni qual volta dovevamo aggiungere un ID di un documento ad uno dei bucket. C'era il rischio che l'aggiunta di un nuovo ID venisse sovrascritta da un altro thread. Una volta che un thread ha finito l'hashing nei buckets, abbiamo usato una barriera per assicurarci che tutti i thread avessero finito l'hashing. Dopodichè, per ogni bucket, abbiamo preso gli ID dei documenti a coppie e verificato se non siano già stati confrontati. Anche qui abbiamo usato un mutex per assicurarci che questo controllo non fosse soggetto a una corsa critica.

SCELTE IMPLEMENTATIVE OPENMP

Abbiamo deciso di parallelizzare quasi ogni cosa che richiedesse almeno un ciclo for, tramite l'utilizzo della pragma "pragma omp parallel for".

Abbiamo avuto bisogno di un primo lock per proteggere una corsa critica nell'LSH: ogni thread effettua l'hash delle sue bande (assegnate in base al rank) in un bucket. Per tale ragione è possibile che due thread contemporaneamente effettuino l'hash della banda nello stesso bucket, dunque l'aggiornamento della matrice dei bucket è una corsa critica.

Abbiamo avuto bisogno di un secondo lock per proteggere un'altra corsa critica nell'LSH, per confrontare due documenti nello stesso bucket. Infatti, nonostante ogni thread confronti i propri bucket, è possibile che due documenti collidano in due bucket diversi. Dunque due thread distinti potrebbero voler confrontare la stessa coppia di documenti.

Abbiamo scelto di usare lo schedule “guided” durante il confronto dei documenti nei bucket in quanto si può verificare che ad un thread (con lo schedule di default, cioè static) vengano assegnati solamente dei bucket in cui ci sono pochissime collisioni (se non nessuna).

PREREQUISITI

Requisiti minimi per eseguire il codice

Cmake v3.2 (minimo)

Almeno C11 (in quanto abbiamo utilizzato l’implementazione delle barriere in pthread)

Come eseguire il codice

Assicurarsi di stare nella working directory del codice. Da linea di comando passare in ordine:

- Nome della cartella contenente i file (una cartella unica)
- Dimensione degli shingles
- Numero di funzioni di hash
- Numero di righe per banda
- Numero di buckets
- Numero di thread (solo per i programmi)

Abbiamo assunto che il numero di thread utilizzati divida sempre il numero di documenti e il numero di funzioni hash.

PRESTAZIONI

I parametri che abbiamo utilizzato sono:

- Dimensione degli shingles: 9
- Numero di funzioni di hash: 200
- Numero di righe per banda: 5
- Numero di buckets: 5000

Abbiamo testato le prestazioni dei nostri programmi con i seguenti valori:

- Dimensioni del problema: 10, 50, 100, 1000
- Numero thread : 1, 2, 5, 10

SERIALE

Numero Documenti	10 documenti	50 documenti	100 documenti	1000 documenti
Seriale	0.43 s	9.9 s	42.5 s	nonDisponibile

Tempi in secondi del programma seriale

PTHREAD

	10 documenti	50 documenti	100 documenti	1000 documenti
1 thread	0.46 s	10.5 s	42.1 s	nonDisponibile
2 thread	0.23 s	5.15 s	20.6 s	nonDisponibile
5 thread	0.11 s	2.63 s	10.8 s	nonDisponibile
10 thread	0.11 s	2.11 s	8.67 s	1134.5 s

Tempi in secondi del programma parallelo implementato con Pthread

SPEEDUP

	10 documenti	50 documenti	100 documenti	1000 documenti
1 thread	1.0	1.0	1.0	nonDisponibile
2 thread	1.87	1.92	2.06	nonDisponibile
5 thread	3.9	3.76	3.93	nonDisponibile
10 thread	3.9	4.69	4.9	nonDisponibile

EFFICIENZA

	10 documenti	50 documenti	100 documenti	1000 documenti
1 thread	1	1	1	nonDisponibile
2 thread	0.935	0.96	1.0	nonDisponibile
5 thread	0.78	0.752	0.786	nonDisponibile
10 thread	0.39	0.47	0.49	nonDisponibile

OPENMP

	10 documenti	50 documenti	100 documenti	1000 documenti
1 thread	0.1 s	10.2 s	42.6 s	nonDisponibile
2 thread	0.26 s	5.2 s	21.6 s	nonDisponibile
5 thread	0.14 s	2.5 s	10.4 s	nonDisponibile
10 thread	0.13 s	2.03 s	7.7 s	1017.8 s

Tempi in secondi del programma parallelo implementato con OpenMp

SPEEDUP

	10 documenti	50 documenti	100 documenti	1000 documenti
1 thread	1.0	1.0	1.0	nonDisponibile
2 thread	1.65	1.90	1.97	nonDisponibile
5 thread	3.07	3.96	4.08	nonDisponibile
10 thread	3.3	4.87	5.52	nonDisponibile

EFFICIENZA

	10 documenti	50 documenti	100 documenti	1000 documenti
1 thread	1	1	1	nonDisponibile
2 thread	0.82	0.95	0.98	nonDisponibile
5 thread	0.614	0.79	0.82	nonDisponibile
10 thread	0.33	0.48	0.55	nonDisponibile

RIFERIMENTI

1. http://infolab.stanford.edu/~ullman/mmds/ch3.pdf
2. https://mrhasankthse.github.io/riz/2020/03/19/Minhash-and-LSH.html
3. http://matthewcasperson.blogspot.com/2013/11/minhash-for-dummies.html