Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Analisi e progettazione di algoritmi di data mining streaming per l'analisi online dei dati.

Andrea Giordano,
Giandomenico Spezzano,
Andrea Vinci

RT-ICAR-CS-14-02 **Giugno 2014**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sezione di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.icar.cnr.it*
– Sezione di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.icar.cnr.it*

Sommario

Con il termine data stream si identificano quei flussi di dati, supposti di dimensione infinita, nei quali nuovi elementi si presentano per essere elaborati col passare del tempo, con una frequenza anche molto elevata. In questo documento sono riportati concetti, modelli, algoritmi e tecniche utili per l'elaborazione e l'analisi online di tali flussi.

Indice

1. Introduzione

Sono sempre più numerosi i contesti nei quali vengono continuamente generate grandi moli di dati che necessitano di essere processate ed elaborate in tempi stringenti.

Trattare un input del genere in modo classico comporta alcuni problemi in diverse fasi dell'elaborazione.

Consideriamo un normale workflow di analisi dei dati, che prevede le fasi di:

1. Acquisizione
2. Memorizzazione
3. Elaborazione

I dati da analizzare sono generalmente prodotti e collezionati alla fonte e da lì trasmessi e consolidati in un base di dati, che mantiene e conserva tutto le informazioni per un tempo indefinito. Algoritmi di analisi lavorano su tutto il volume di dati immagazzinato, eventualmente effettuandone più di una scansione, con tempi di elaborazione non immediati ed in modalità "on demand", elaborando a precisa richiesta da parte di un utente.

Una semplice interrogazione SQL, ad esempio, effettuerà le opportune operazioni di join e di selezione (filtraggio) sulle tabelle del database, considerando volta per volta, per ogni interrogazione, tutti dati disponibili. Od ancora, l'algoritmo k-means, un algoritmo di clustering classico, elabora leggendo e rileggendo più volte, una per iterazione, tutti i dati disponibili, al fine di ricalcolare i cluster e migliorare il risultato. Questa modalità di calcolo richiede quindi più scansioni dell'intero input fornito.

Quando i dati sono generati a ritmi molto elevati oppure se si necessitano tempi di risposta rapidi o immediati, l'approccio classico sopra descritto non è più applicabile. L'acquisizione e la trasmissione dei dati possono, ad esempio, produrre ritardi dovuti a carenze della rete o sovraccarichi momentanei. La memorizzazione potrebbe richiedere più capacità di quella disponibile, in quanto, con l'approccio sopra descritto, è necessario conservare e mantenere tutti i dati prodotti per un lungo periodo. Una elaborazione complessa e che consideri e riconsideri più volte tutti i dati disponibili potrebbe impiegare tempi di computazione non in linea con i requisiti.

Se, quindi, si ha a che fare con un contesto nel quale i dati vengono generati molto velocemente e si ha necessità di contenere i tempi di calcolo e lo spazio di memorizzazione, pur sacrificando ragionevolmente la precisione dei risultati, è il caso di cambiare approccio. Si utilizzano quindi modelli di elaborazione nei quali è previsto che un elemento dell'input possa essere letto solo una volta, per poi non essere più disponibile. La computazione è effettuata generalmente aggiornando un risultato parziale man mano che i dati vengono letti, usando magari una sinossi dei dati letti che occupi uno spazio molto inferiore, se non costante, rispetto alla dimensione dell'input. Si passa quindi ad un modello adatto a trattare i flussi di dati di dimensione anche infinita.

Il modello dei Data Streams ben si adatta al contesto di progetto, in quanto si prevede di avere un gran numero di oggetti intelligenti immersi in un ambiente fisico che producono continuamente dati ed attuano comportamenti. I dati provenienti dagli Smart Object, siano essi misure fisiche grezze o aggregazioni, od anche informazioni sullo stato di funzionamento degli oggetti, sono prodotti ad alta velocità, e costituiscono in effetti dei flussi di dati. L'analisi di questi deve essere svolta con opportuni algoritmi che elaborino in tempo reale ed online, per poter rispondere rapidamente alle esigenze del sistema.

Di seguito, nella sezione 2 verranno presentati e formalizzati i concetti alla base dei Data Stream, insieme ad una architettura per la gestione degli stessi ed alle tecniche di campionamento e filtraggio note, utili per alleggerire il carico computazionale sfoltendo opportunamente l'input. Nelle sezioni 3 e 4 si mostreranno alcuni algoritmi in grado di computare grandezze statistiche su flussi di dati, grandezze che possano descrivere qualitativamente i dati per una prima analisi. Nella sezione 5 ci si concentrerà sugli algoritmi di clustering su flussi di dati, che sono alla base di tecniche che consentano di individuare in tempo reale l'evoluzione del sistema e il comportamento collettivo degli oggetti intelligenti presenti.

2. Data streams: Concetti di Base e modelli

Un data stream (flusso di dati) è, formalmente, una sequenza ordinata di elementi $\{a_1, a_2, \dots, a_n\}$, presupposta di lunghezza infinita, che descrive un segnale $\boldsymbol{A}$, funzione $\boldsymbol{A}$: $[1 \dots N] \rightarrow R$.

Vi sono diversi modelli[15] che definiscono in quale modo la sequenza di a_i descrive il segnale A:

1. Modello a Serie Storica: ogni a_i è esattamente uguale ad $A[i]$, ed appare in ordine temporalmente crescente secondo i. Questo modello si applica, ad esempio, quando l'elemento a_i della sequenza rappresenta il volume di traffico su un link IP, osservato ogni 5 minuti.
2. Modello a Registro Cassa: gli a_i sono incrementi per $A[j]$. In questo caso, lo stato del segnale $A_i[j]$ rappresenta lo stato dopo ad aver osservato la sequenza fino all'i-esimo elemento. Se un elemento della sequenza a_i rappresenta un accesso ad una pagina web, ad esempio, il segnale $A[j]$ rappresenta la somma di tutti gli accessi intercorsi dal calcolo di $A[j-1]$. In questo modello gli a_i sono considerati positivi.
3. Modello a Tornello: come il precedente, soltanto che gli elementi della sequenza possono essere negativi.
4. Modello a "Finestre": In alcuni contesti, soprattutto quando si vogliono eseguire task di data mining su basi di dati, i dati più recenti sono considerati maggiormente rilevanti rispetto a quelli passati. Quando si vuole monitorare un sistema nella sua evoluzione per, ad esempio, riconoscere l'avvenire di un certo comportamento od una situazione di rischio, si da maggior peso ai dati recenti, che rappresentano lo stato del sistema nel momento nel quale lo si sta osservando. In questo caso lo stato del segnale $A[j]$ dipende in modo maggiore dagli elementi più recenti della sequenza a_i, ed in modo minore da quelli passati. Due sono gli approcci possibili
    a. Combinatorio o a finestre scorrevoli: si considerano, per la computazione dello stato $A[j]$, soltanto gli ultimi w elementi osservati nella sequenza a_i, dove w è la dimensione della finestra. Gli elementi in eccesso sono semplicemente scartati.
    b. Telescopico o finestre smorzate o time-fading o decaying windows: si utilizza un secondo segnale $B[j]$, rappresentante lo stato interno del segnale principale $A[j]$, in modo simile a quanto utilizzato nei modelli di sistemi dinamici. Lo stato $B[j]$ dipenderà da $B[j-1]$, attenuato di un fattoreλ, e dagli ultimi w elementi di a_i, mentre $A[j]$ dipenderà da $B[j]$ e dagli ultimi w elementi di a_i.

Data-Stream-Management System

In [11] è definita una architettura per un sistema di gestione dei flussi di dati. Come per i database management systems, è possibile pensare un'architettura per data-stream management system come in figura 1. In ingresso al sistema si possono avere un qualsiasi numero di flussi, ognuno dei quali può fornire dati indipendentemente, con velocità diverse e tipi di dato diverse. I flussi possono essere memorizzati integralmente in un *archivio compresso*, utilizzabile per analisi storiche offline.

È disponibile uno spazio di memoria limitato (*limited working storage)* per l'elaborazione dei flussi, contenente generalmente soltanto porzioni dei flussi dati o strutture utili per costruire

un modello sintetico ed approssimato dei flussi stessi. Questo spazio di memoria può risiedere sul disco o in memoria principale, ma è comunque definito piccolo rispetto alla dimensione (anche infinita) degli elementi di un flusso.

Lo Stream Processor elabora i flussi in ingresso e produce dei flussi in uscita, rispondendo a delle query ad-hoc. Lo stream processor può utilizzare il *working storage* per l'elaborazione, nelle modalità appena descritte.

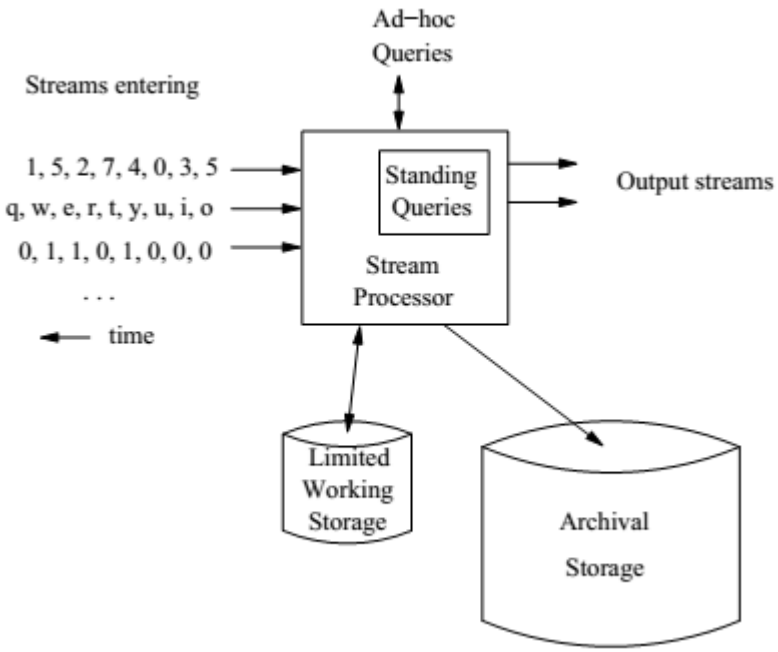

Figura 1. Architettura di un Data Stream Management System.

Prima di elaborare i flussi, è generalmente necessario ridurre la velocità di generazione ed acquisizione dei dati, che potrebbe essere troppo elevata per poter essere processata in tempo reale, oppure è possibile che un flusso dati contenga, insieme agli elementi utili per una certa analisi, anche elementi non di interesse. Per risolvere queste problematiche sono necessarie tecniche di campionamento e filtraggio rapide, due delle quali sono descritte qui di seguito.

Campionamento dei flussi dati.

In molti casi d'uso reali ed in molti algoritmi per data stream, il rateo di produzione dei dati è tanto elevato che è necessario comunque eseguire un campionamento del flusso dati d'input. Spesso, la scelta del metodo di campionamento più corretto è risolutiva per la corretta ed efficiente esecuzione dell'algoritmo di analisi.

Campionare vuol dire, in soldoni, scegliere accuratamente (e velocemente) la sotto-porzione dei dati statisticamente più rappresentativa per poter eseguire il compito richiesto. Raramente la scelta del metodo di campionamento corretto si risolve con l'essere quello più banale.

Consideriamo, ad esempio, di avere un flusso di dati in input, nel quale ogni elemento è una tripla <utente, interrogazione, tempo>, rappresentante le interrogazioni richieste da un certo utente ad un motore di ricerca, e di voler rispondere alla domanda: "Qual è la frazione tipica di interrogazioni ripetute dagli utenti nell'ultimo mese?". Dobbiamo calcolare il rapporto tra il numero di interrogazioni ripetute rispetto al totale delle interrogazioni per ogni utente, e farne una media. Consideriamo anche di volere occupare 1/10 dello spazio richiesto per mantenere l'intero flusso.

Senza campionare, la risposta corretta si otterrebbe contando, per ogni utente, il numero s di interrogazioni singole (cioè effettuate una sola volta) ed il numero d di interrogazioni almeno doppie (cioè effettuate almeno due volte). La risposta sarebbe quindi pari a $d/(s+d)$.

Se si effettuasse un campionamento banale, ad esempio scegliendo di mantenere una tripla a caso ogni 10, ridurremmo lo spazio necessario per la computazione ad 1/10, come richiesto, ma otterremmo un grosso errore sul risultato. Campionando in questo modo, infatti, considerando di avere 1/10 dei dati, conteremmo in effetti s/10 interrogazioni singole per ogni utente, ma delle d interrogazioni ripetute 2 volte, soltanto d/100 comparirebbero come tali nell'insieme campionato (cioè la probabilità che entrambe le ripetizioni siano scelte casualmente come campioni), portando ad un grosso errore nel risultato finale.

Una soluzione più furba consiste nel campionare non in modo completamente casuale, ma, ad esempio, scegliendo 1/10 della base di utenti. In questo caso manterremo comunque i vincoli di occupazione spaziale, poichè ad 1/10 della base di utenti corrisponderà grossomodo 1/10 delle tuple totali, ma saremo in grado di ottenere un risultato più corretto poiché non avremo la distorsione introdotta nel caso precedente. Il campionamento può essere effettuato semplicemente realizzando una funzione di hash che associa ad ogni nome utente un numero tra 0 e 9, selezionando le tuple che aventi per utente un hash corrispondente pari a 0 e scartando le restanti.

In generale, quindi, supponendo di avere un flusso dati composto da tuple, per poter campionare correttamente è necessario selezionare quali attributi siano da considerarsi chiavi per il risultato che si vuole ottenere, e campionare considerando i valori di quegli attributi, mediante una opportuna funzione di hash.

Abbiamo considerato, nell'esempio, un vincolo di occupazione in memoria lineare rispetto alla dimensione dell'input. È possibile rendere questo vincolo ancora più stringente. Consideriamo di avere fissato il numero di tuple che è possibile tenere in memoria, ed in input un flusso continuo infinito, la cui dimensione cresce, quindi, nel tempo. Consideriamo di utilizzare una funzione di hash che associa alle chiavi individuate dei valori in un range tra 1 ed un certo valore B-1 elevato. Abbiamo quindi B possibili valori di hash.

Consideriamo anche t valore di soglia che diminuisca nel tempo, inizialmente pari al numero B di possibili valori della funzione di hash. A mano a mano che i dati si presentano in input, il campione scelto sarà formato dalle tuple le cui chiavi k soddisfano: $hash(k) <= t$. Nuove tuple saranno aggiunte al campione soltanto se il loro hash è inferiore alla soglia, ed anche le vecchie

tuple che non soddisfano più la precedente condizione verranno scartate, liberando spazio per i nuovi campioni.

Filtraggio dei flussi dati e filtro di Bloom.

Se il flusso in input presenta sia elementi che sono di interesse per l'analisi, distinguibili per caratteristiche ben specificate, sia elementi che invece non sono affatto utili, si possono utilizzare processi di filtraggio che selezionino e presentino all'algoritmo di mining soltanto gli elementi di interesse, scartando velocemente le informazioni irrilevanti.

Filtrare un flusso dati, in questo contesto, vuol dire accettare gli elementi del flusso che soddisfano uno specifico criterio, e scartare tutti gli altri. Se il criterio di selezione è rappresentato da una funzione dipendente solamente dall'elemento stesso, il filtraggio è un processo piuttosto semplice.

Supponiamo, ad esempio, di avere un flusso di tuple del tipo <rumore, coordinate, tempo>, rappresentanti ognuna una misurazione di un sensore di rumore, eseguita in certe coordinate gps in un certo istante di tempo. Per il nostro task di mining sono di interesse soltanto gli elementi del flusso che presentano un valore dell'attributo rumore maggiore di 50 decibel. In questo caso, scegliere se l'elemento deve essere o meno accettato è un processo banale, in quanto richiede solo una comparazione dell'attributo *rumore* ad una soglia specificata (50 db).

In altri casi, però, il criterio di accettazione potrebbe non dipendere soltanto dalle proprietà dello specifico elemento. Poniamo, ad esempio, di dover analizzare un flusso di email, rappresentate da tuple del tipo <mittente, destinatario, corpo>, e supponiamo ancora, molto realisticamente, che in questo flusso siano presenti numerose email di spam, prive di interesse per il nostro task di mining. Vogliamo accettare soltanto email provenienti da utenti ben specificati, appartenenti ad una whitelist, che supponiamo possa avere anche un miliardo di indirizzi consentiti. In questo caso, valutare l'accettazione od il rigetto di un elemento è non banale, perché necessità di verificare l'appartenenza dell'elemento ad un certo insieme, di dimensione troppo grande per pensare possa essere mantenuto in memoria principale.

Una possibile soluzione a questo problema consiste nell'applicare un filtro di Bloom [12], specificatamente pensato per, ed adatto a, questo contesto. Un filtro di Bloom è una struttura efficiente in spazio che consente di testare se un elemento appartiene o non appartiene ad un insieme, con risultati che ammettono falsi positivi, ma non falsi negativi. Il filtro di Bloom ha quindi una recall pari al 100%.

Un filtro di Bloom consiste di:

1. Un array B di n bits, inizializzato a 0.
2. Una collezione di funzioni di hash $h1, \ldots, hk$. Ogni funzione mappa una *chiave* in un array di n bit.
3. Un insieme S di m chiavi consentite.

Scopo del filtro è quello di accettare tutte le chiavi presenti nell'insieme S, scartando la gran parte delle chiavi non consentite. Per inizializzare il filtro di Bloom a partire dall'array B

inizializzato con ogni elemento a 0, si prende ogni chiave presente in *S* e si calcolano per quella chiave tutte le *k* funzioni di hash. Quindi si aggiorna l'array *B* inserendo il valore 1 per i bit corrispondenti ad ognuna dei valori hash così calcolati (Figura 2).

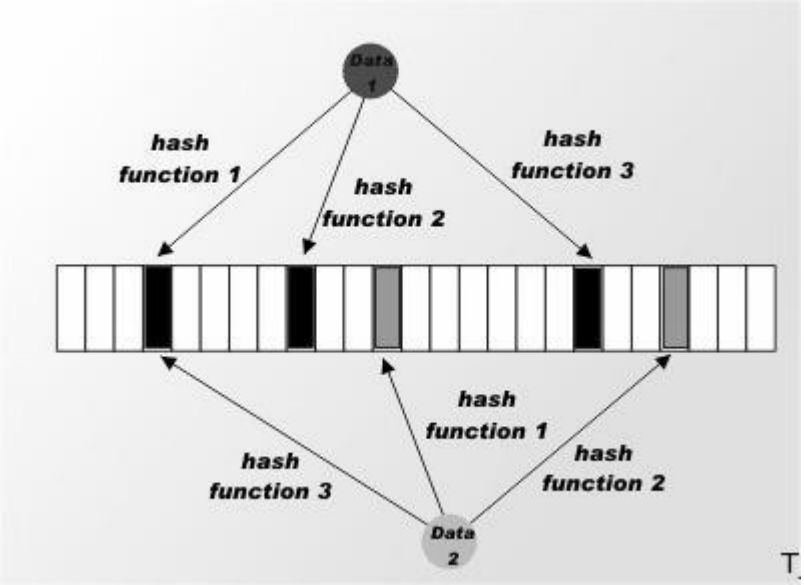

Figura 2. Inizializzazione del filtro di Bloom con due chiavi (data 1 e data 2) e tre funzioni di hash.

Dopo aver inizializzato il filtro, per testare se una chiave appartiene o meno all'insieme *S*, si calcolano tutte le funzioni di hash per quella chiave e si verifica che ai bit 1 di ogni hash corrispondano, nella stessa posizione, bit 1 nell'array B. Se per ogni chiave ed ogni bit 1 esiste questa corrispondenza, l'elemento è accettato, altrimenti è scartato.

È chiaro che, se un elemento è scartato, la sua chiave sicuramente non appartiene al set di chiavi consentite. Infatti, se una chiave è scartata vuol dire che ad almeno un bit 1 di almeno un suo valore di hash corrisponde ad un bit 0 in *B.* Se la chiave testata appartenesse ad *S,* ciò sarebbe impossibile per costruzione del filtro di Bloom.

Se un elemento è accettato, però, potrebbe comunque non essere presente dell'insieme *S*, in quanto potrebbe verificarsi la corrispondenza di tutti i bit 1 dei sui hash con i bit 1 di *B* inseriti da un gruppo di altre chiavi consentite. Per minimizzare la probabilità di falso positivo, le funzioni di hash devono essere uniformemente distribuite e tra loro indipendenti. In questo caso, si dimostra che la probabilità di falso positivo è pari a: $\left(1 - e^{-\frac{kn}{m}}\right)^k$, con *n* numero di bit di *B*, *k* numero di funzioni di hash, *m* numero di chiavi in *S*. Questo vuol dire che, se avessimo un filtro di Bloom con *B* di 1 megabyte, utilizzando 5 funzioni di hash potremmo considerare un insieme S di 1 milione di elementi, con una probabilità di falso positivo inferiore al 2,5%.

3. Algoritmi su Data stream: stime statistiche

Dopo aver brevemente mostrato cosa sia un flusso dati ed introdotto una possibile architettura, insieme alle nozioni di campionamento e filtraggio, di seguito verranno mostrati alcuni algoritmi utili ad eseguire una analisi statistica degli elementi di un flusso dati. Mostreremo quindi tecniche per stimare momenti di diverso ordine per un flusso dati.

Stima dei momenti

Calcolare i momenti di un flusso dati vuol dire ottenere informazioni qualitative sulla distribuzione delle frequenze dei differenti elementi del flusso.

Supponiamo di avere un flusso dati che consta di elementi appartenenti ad un certo insieme degli elementi possibili, e che questi possano essere ordinati, in modo che sia possibile riferirsi al i-esimo elemento dell'insieme. Poniamo m_i come il numero di volte che l'elemento i compare nel flusso dati, il *momento di ordine k* del flusso dati sarà pari a $\sum_i (m_i)^k$.

Il momento di ordine zero ($k = 0$) rappresenta quindi il numero degli elementi del flusso distinti (maggiori di 0). Il momento del primo ordine ($k = 1$) è pari alla somma di tutte le frequenze m_i, cioè è pari alla dimensione del flusso.

Il momento del secondo ordine è la somma dei quadrati degli m_i. Questo rappresenta quanto la distribuzione del flusso sia non uniforme. Ricorriamo ad un esempio per chiarire l'idea: supponiamo di avere un flusso di 100 elementi, composto da una serie di 10 possibili elementi, e che ogni elemento compaia 10 volte (distribuzione uniforme). In questo caso, il momento del secondo ordine sarà pari a $10 \times 10^2 = 1000$. Se invece la distribuzione degli elementi non è uniforme, il momento del secondo ordine sarà maggiore. Considerando ad esempio che un elemento compaia 37 volte e che ogni altro elemento compaia 7 volte, il momento del secondo ordine sarà molto maggiore rispetto al caso precedente: $1 \times 37^2 + 9 \times 7^2 = 1810$.

Dai momenti di diverso ordine così definiti è calcolare importanti grandezze statistiche, quali l'indice di Gini [14].

Mentre il calcolo del momento del primo ordine di un flusso dati è banale, in quanto basta contare il numero degli elementi nel flusso via via che si presentano, calcolare o stimare il momento di ordine 0 (il numero di elementi distinti) ed il momento del secondo ordine risulta essere più complesso.

Stima del momento di ordine 0: elementi distinti in un flusso ed algoritmo di Flajolet-Martin

Si è già descritto come al momento di ordine 0 corrisponda il conteggio degli elementi distinti nel flusso. Per effettuare questo conteggio esattamente, sarebbe necessario mantenere in memoria principale una struttura che contenga l'insieme degli elementi distinti già arrivati ed un contatore. Quando si legge un nuovo elemento dal flusso, si verifica che esso non sia presente nella struttura, in caso positivo lo si aggiunge e si incrementa il contatore, nel caso negativo lo si scarta. Questa soluzione comporterebbe una occupazione in spazio pari a tutti i possibili

elementi che possano appartenere al flusso il che è spesso improponibile (consideriamo, ad esempio un flusso di stringhe immesse in un motore di ricerca).

Una soluzione approssimata, basata ancora una volta su funzioni di hash, è rappresentata dall'algoritmo di Flajolet-Martin [1][2][3]. Esso è in grado di stimare in una sola scansione dell'input (una sola lettura di ciascun elemento) gli elementi unici in un flusso, con complessità spaziale di $O(\log(m))$, dove m è pari al numero di elementi unici (l'algoritmo sopra introdotto aveva, invece, complessità lineare $O(m)$). Questo algoritmo fornisce una stima approssimata, ma anche la deviazione standard di quella stima, che può essere utilizzata per limitare l'approssimazione ad un certo errore ε.

L'intuizione alla base di questa tecnica è che, se si utilizza una buona funzione di hashing che opera sugli elementi e genera numeri interi secondo una distribuzione uniforme, possono essere fatte alcune considerazioni sulle chiavi intere binarie generate da un insieme di elementi:

- La metà delle chiavi binarie terminerà in con uno 0 (è cioè divisibile per 2)
- $\frac{1}{4}$ delle chiavi terminerà in 00, (è cioè divisibile per 4)
- $\frac{1}{8}$ delle chiavi terminerà in 000, (divisibile per 4
- …
- In generale, $\frac{1}{2^n}$ chiavi binarie termineranno con n 0 (0^n).

Di conseguenza, se la funzione hash applicata agli elementi del flusso ha generato un intero terminante in 0^m elementi (e chiavi terminanti in $i-1$ bit 0, $i-2$ bit 0, etc), allora il numero di elementi unici sarà nell'ordine di 2^m.

Una implementazione dell'algoritmo di Flajolet-Martin consta dei seguenti passi:

1. Inizializza a 0 un array B di bit di dimensione L tale che $2^L > n$, con n numero di elementi nel flusso. Con già $L = 64\ bit$ è possibile contare gli elementi distinti in un flusso di URL web. L'$i-esimo$ bit di questo array assumerà valore 1 se nella scansione degli elementi si è ne è già incontrato con la cui funzione di hash terminava con i 0.
2. Si genera una funzione di hash che mappa gli elementi del flusso in interi, lunghi L bit. Questa funzione deve generare interi uniformemente distribuiti.
3. Si legge un elemento dell'input, e se ne calcola la funzione di hash. Si valuta la "coda" di 0 dell'intero calcolato: se termina in i zeri, si setta ad 1 l'i-esimo bit dell'array B, quindi si legge l'elemento successivo.

In qualunque momento, la stima del numero degli elementi distinti del flusso letto sarà pari a 2^R, con R pari alla posizione del primo 0 nell'array.

La stima calcolata in questo modo sarà sempre una potenza di 2. Si può ovviare a questo, e produrre una stima più precisa, se si utilizza un insieme di funzioni di hash (associate ognuna ad un suo array di valori), calcolando le diverse stime ed aggregandole mediante operazioni di media o mediana, o combinazioni delle due. La metodologia più corretta in letteratura prevede l'utilizzo di un insieme funzioni hash, partizionate in sotto-insieme. La stima di ogni sotto-

insieme è calcolata come la media delle stime dei suoi elementi, mentre la stima totale è calcolata come mediana delle stime dei sottoinsiemi. In questo modo si riduce notevolmente l'errore.

Stima del momento di secondo ordine: algoritmo di Alon-Matias-Szegedy

Per calcolare esattamente il momento di secondo ordine di un flusso di dati, sarebbe necessario, come nel caso precedente, il mantenimento in memoria principale di una struttura contenente, per ogni possibile elemento del flusso, il conteggio del numero di volte in cui esso è stato letto dall'input. Lo spazio necessario, in tal modo, sarebbe nuovamente lineare rispetto alla dimensione dell'input.

Per la stima del momento del secondo ordine, una soluzione è stata proposta da Alon, Matias e Szegdy [3][4]. Questa soluzione comporta l'utilizzo di un approccio probabilistico basato sull'utilizzo di un certo numero di variabili $X = (elemento, valore)$ tali che:

- $X.elemento$ sia uno dei possibili elementi nel flusso;
- $X.valore$ conta il numero degli $X.elemento$ nel flusso dati, a partire da una posizione scelta casualmente. Quindi, a partire di una certa posizione del flusso, ogni volta che l'elemento $X.elemento$ verrà letto in input, $X.valore$ sarà incrementato.

La stima del momento di secondo ordine per ogni variabile X sarà pari a $n \times (2 \times X.valore - 1)$. La dimostrazione della validità di questo stimatore si trova in [4].

Aggregando le stime fornite da più variabili X_i attraverso l'operatore di media, riusciamo ad ottenere stime del momento di secondo ordine via via migliori. Quante più variabili saranno utilizzate, maggiore sarà l'occupazione in spazio, migliore sarà la stima.

4. Algoritmi su data stream per il modello a finestre

Gli algoritmi di Flajolet-Martin e Alon-Matias-Szegedy presentati brevemente nei paragrafi precedenti presupponevano un modello di flusso dati del tipo "a serie storica", nei quali si voleva computare gli stimatori dei momenti di diverso ordine su tutto il flusso dati, dal primo elemento letto, fino all'ultimo elemento acquisito.

Di seguito presentiamo due algoritmi per l'analisi di flussi di dati basati su un modello a finestre. Il primo risolve il problema del conteggio degli elementi in una sliding window, il secondo quello del conteggio dell'elemento più frequente utilizzando il modello delle decaying window.

Contare gli Uno in una finestra scorrevole di bit: l'algoritmo di Datar-Gionis-Indyk-Motwani

L'algoritmo vuole rispondere al problema: "quanti 1 sono presenti negli ultimi k bit di un flusso dati?". Si utilizza quindi un modello a finestre scorrevoli, nel quale si vuole una risposta che consideri solamente gli ultimi k elementi di un flusso. L'algoritmo può essere esteso per calcolare altre grandezze statistiche di rilievo quale, ad esempio, la sommatoria degli ultimi k elementi in un flusso di interi, o la media.

Supponiamo quindi di avere un flusso infinito di elementi, che ogni elemento sia un bit, e che possa valere 0 oppure 1. Contare esattamente il numero di bit 1 degli ultimi k bit letti comporterebbe il mantenere in memoria una coda contenente k elementi (inizializzata a 0) ed un contatore. Ogni qual volta un nuovo elemento del flusso viene processato, viene inserito in testa alla coda, mentre viene rimosso e scartato l'ultimo elemento della struttura, aggiornando opportunamente il contatore. Questo algoritmo occupa uno spazio lineare in k, necessario per rappresentare l'intera finestra.

L'algoritmo di Datar-Gionis-Indyk-Motwani (DGIM) [5] riesce invece a rappresentare una finestra utilizzando soltanto $O(\frac{1}{\varepsilon}\log^2 k)$ bit, stimando il numero di 1 nella finestra con un errore ε. Si è quindi in grado di ridurre l'errore pagando un costo nello spazio occupato.

Consideriamo di associare ad ogni bit del flusso dati un *timestamp,* che ne rappresenti la posizione nel flusso. Il primo bit del flusso avrà timestamp pari ad 1, il secondo pari a 2, etc. Poiché consideriamo una finestra di k elementi, è possibile rappresentare i timestamp in modulo k, ciò consente di utilizzare solo $\log_2 k$ bit per descrivere un singolo timestamp.

Si divide la finestra in diversi *bucket (porzioni),* ognuno formato da:

1. Il timestamp del suo ultimo elemento
2. Il numero di 1 nel bucket. Questo numero deve essere una potenza di 2, ed è chiamato *dimensione* del bucket.

Per rappresentare un bucket sono necessari $\log_2 k$ bit per il timestamp, e $\log_2 \log_2 k$ bit per la dimensione, considerando che debba essere una potenza di 2. Quindi sono necessari soltanto $O(\log k)$ bit per rappresentare un bucket.

Affinchè un flusso sia rappresentabile come sequenza di bucket, deve essere:

1. L'ultimo elemento di un bucket deve essere una posizione contenente un 1.
2. Ogni posizione del flusso contenente un 1 deve essere in un qualche bucket.
3. Nessuna posizione può essere presente in più bucket.
4. Ci sono uno od al più due bucket con la stessa dimensione, fino ad una dimensione massima definita.
5. Tutti i bucket devono avere dimensione pari ad una potenza di 2.
6. La dimensione dei bucket non può diminuire con lo scorrere della finestra.

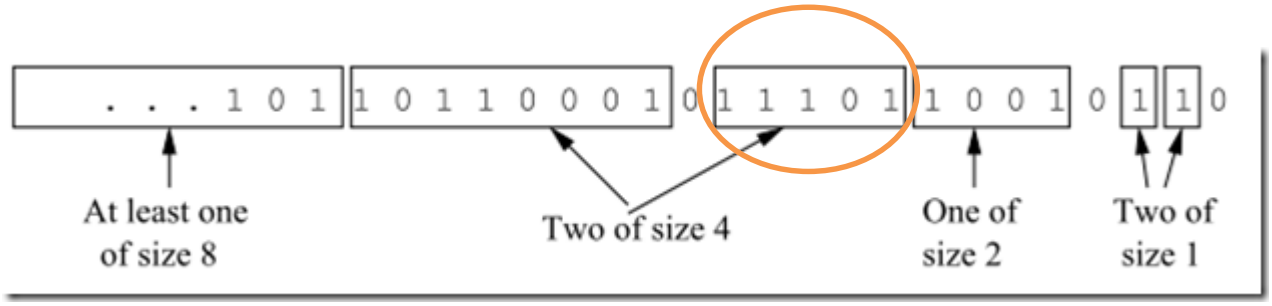

Figura 3. Un esempio di una finestra divisi in bucket secondo le precedenti regole.

Per stimare quanti 1 vi sono in una finestra di un flusso così formato, l'algoritmo DGIM prevede che si trovi il bucket b meno recente che contenga informazioni su almeno un bit della finestra considerata, e che si calcoli la somma delle dimensioni di tutti i bucket più recenti presenti nella finestra, più la metà della dimensione di b.

Se, ad esempio, consideriamo la porzione di flusso in figura, e la stessa suddivisione in bucket, considerando una finestra formata dai $k = 10$ elementi più recenti (più a destra). Il bucket b scelto sarà quello cerchiato, poiché è l'ultimo che contiene un qualche elemento della finestra, è la stima del numero di 1 sarà data dalla somma delle dimensioni dei bucket più recenti di b, cioè alla sua destra, più metà della sua dimensione, e quindi: $1 + 1 + 2 + \frac{4}{2} = 6$.

Utlizzando questa stima, l'errore massimo atteso è del 50%, dipendente dall'approssimazione sul bucket scelto b, che è quello con dimensione maggiore nella finestra. In [5] si mostra come questo errore possa essere notevolmente ridotto.

La struttura a bucket deve essere mantenuta ed aggiornata man mano che i dati sono letti dal flusso. Per fare questo, ogni volta che un nuovo bit è letto:

- Si verifica che il bucket meno recente rappresenti ancora un qualche elemento della finestra. Se questo non è, lo si scarta.
- Se il nuovo elemento è un bit 1:
    - Si crea un bucket con il timestamp corrente e dimensione 1.
    - Se vi sono tre bucket con dimensione 1, i due meno recenti vengono aggregati in uno di dimensione doppia. Il timestamp associato al nuovo bucket è quello del bucket più recente tra quelli che aggrega. Se dopo questa aggregazione vi sono ancora più di 2 bucket con la stessa dimensione, anche quelli si aggregano, e via di seguito, finchè non vi sono al più due bucket con la stessa dimensione.

In [5] sono mostrate estensioni di questo approccio per la stima di altre grandezze statistiche.

Modello telescopico o decaying windows: stima dell'elemento più frequente.

Mentre il modello a finestre scorrevoli (sliding windows) considera gli ultimi k elementi di un flusso come egualmente importanti, e tutti gli altri elementi come insignificanti, il modello a decaying windows (finestre smorzate, o modello telescopico) considera una transizione più *morbida* tra elementi recenti e passati. In questo modello, infatti, gli elementi perdono importanza in modo graduale man mano che nuovi elementi si presentano in input. Questo avviene introducendo una nozione di *peso* associata al timestamp degli elementi, che decresca, generalmente in modo esponenziale, rispetto al timestamp corrente, ovvero quello dell'ultimo elemento letto.

Consideriamo una sequenza di bit $a_1, a_2, \dots, a_t$, dove a_1è il primo elemento del flusso, ed a_t l'elemento più recente (l'ultimo letto). Stimare gli 1 in un modello a decaying windows comporterebbe calcolare:

$$\sum_{i=0}^{t-1} a_{t-i}(1-c)^i$$

Con c una costante dimensionata secondo quanto velocemente si voglia rendere meno importanti gli elementi meno recenti.

L'utilizzo di questo modello è più semplice rispetto al modello sliding windows, poiché non è necessario mantenere in memoria una intera finestra di elementi fissata. Inoltre, questa funzione si può calcolare in modo incrementale: se un nuovo elemento a_{t+1} si presenta nel flusso dati, basterà moltiplicare la stima precedente per $(1-c)$ ed aggiungere a_{t+1}.

Questo modello può essere utilizzato, ad esempio, per stimare quale sia l'elemento più frequente in un flusso, qualora sia importante privilegiare la storia recente rispetto a quella passata. Consideriamo di avere un flusso di dati in cui ogni elemento rappresenti la vendita di un biglietto per un certo film. Possiamo stimare quale sia il film più popolare al momento mantenendo una variabile per ogni film, e aggiornandone il valore secondo la funzione precedente ogni qual volta un elemento del flusso ne indica l'acquisto.

È possibile anche non mantenere le variabili per ogni film, ma soltanto per quelli più frequenti, semplicemente scegliendo una soglia e scartando, via via, le variabili che siano di valore inferiore a quella soglia, cioè, all'arrivo di un nuovo elemento:

1. Si moltiplica ognuna delle variabili mantenute al momento per $(1-c)$
2. Se l'elemento è un film che non ha una variabile associate, ne si crea una e la si inizializza a 1. Altrimenti, se l'elemento è associato ad una variabile già presente, si incrementa quella variabile di 1-
3. Tutte le variabili sotto il valore di soglia vengono rimosse.

In questo modo, si riducono notevolmente il numero di variabili necessarie per stimare gli elementi più frequenti in un flusso, utilizzando un modello a finsetre smorzate.

5. Data stream clustering

Dopo aver presentato alcuni algoritmi per l'analisi dei flussi di dati in grado di stimare alcune grandezze statistiche con spazio limitato, presentiamo di seguito i principali algoritmi utilizzati per effettuare l'analisi dei cluster di flussi dati.

Fare clustering su un insieme di dati, in questo caso su un flusso, vuol dire partizionare l'insieme in input in sotto-gruppi, tali che tutti li elementi di una partizione siano tra loro massimamente simili, e che ognuno sia massimamente dissimile dagli elementi appartenenti alle altre partizioni.

L'analisi dei cluster è utilizzata in diversi contesti, come l'analisi di sequenze di geni, la chimica industriale, la climatologia, e l'analisi di reti sociali.

Individuare cluster in un flusso dati è un compito non facile, in quanto molti degli algoritmi tradizionali, quali k-means o DB-scan, si adattano molto poco ad un contesto nel quale si ha a che fare con un insieme di dimensione infinita e con il vincolo di non poter leggere i dati in ingresso più di una volta.

Per ovviare a queste problematiche, gli algoritmi presentati qui di seguito comportano, generalmente, l'utilizzo di una struttura sinottica dello storico dei dati letti dal flusso, che sia di dimensione molto piccola, sulla quale poi saranno eseguite varianti degli algoritmi di clustering più tradizionali e noti in letteratura.

STREAM

STREAM[6][7][13] è un algoritmo per il clustering di flussi di dati basato su k-median.

K-median è una variante del K-means, che sfrutta il concetto di medioide piuttosto che quello di centroide. Dato un dataset, un insieme di k elementi del dataset, detti medioidi, che individuano ciascuno un diverso cluster, ed una funzione di similarità definita sugli elementi del dataset, una iterazione del K-median consta di due fasi:

1. Ogni elemento del dataset è assegnato al medioide ad esso più vicino, secondo la funzione di similarità. L'insieme dei punti assegnato allo stesso medioide rappresenta un cluster centrato in quel medioide.
2. Per ogni cluster individuato, se ne ricalcola il medioide come l'elemento del cluster che sia di minima dissimilarità rispetto a tutti gli altri elementi del suo cluster.

L'algoritmo itera fino a raggiungere una soluzione stabile, ovvero quando nessuna modifica dei medioidi è effettuata rispetto alla iterazione precedente o la modifica non ha prodotto un miglioramento della qualità dei cluster, o dopo un numero prefissato di iterazioni.

L'algoritmo STREAM considera di suddividere il flusso dati D in porzioni, $D_1, \dots, D_r, \dots$ ognuna contenente al più m elementi. m è predefinito a seconda del budget di memoria previsto. Ogni qual volta una porzione D_i del flusso è letta in input, si seleziono un sottoinsieme S_i di k-elementi rappresentativi della porzione di dati, con un qualche algoritmo che calcoli k-median su D_i, e si assegna ad ognuno elemento di S_i un "peso", pari al numero di elementi del cluster

che rappresenta. Si procede così per ognuna delle porzioni in arrivo. Quando l'insieme di tutti i rappresentativi di ogni porzione del flusso è maggiore di m, si esegue sull'insieme di tutti i rappresentativi una variante del k-medoids che consideri anche i pesi di ogni punto. In questo modo si ottiene un secondo livello di rappresentativi. In generale , quando rappresentativi di livello p sono in numero maggiori di m, essi sono convertiti in k rappresentativi di ordine $p + 1$.

Questa struttura (l'insieme dei rappresentativi) costituisce una sinossi del flusso. In qualsiasi momento si vogliano computare i cluster sul flusso letto in input, basterà calcolare il k-medoids sull'insieme di tutti gli elementi della sinossi.

CluStream

In alcuni contesti è necessario tracciare l'evoluzione dei cluster di un flusso in funzione del tempo. Si vuole cioè rendere possibile all'analista visualizzare i cluster considerando il momento corrente insieme con i cluster calcolati ad certo momento passato del flusso, per poterli confrontare ed analizzarne l'evoluzione.

L'algoritmo CluStream [7] risponde a questa esigenza elaborando in due fasi: una fase online, in cui si mantiene una sinossi del flusso, aggiornata via via che nuovi elementi si presentano in input, ed una fase offline ed on-demand, nella quale si richiede il calcolo effettivo dei cluster, calcolo che verrà effettuato sulla struttura sinottica mantenuta nella prima fase.

La struttura sinottica è basata su due costrutti fondamentali:

- I *microcluster*, rappresentano l'unità informativa della sinossi, contengono informazioni sulla località dei dati in termini di densità attorno ad una certa aera nello spazio dei dati.
- *Time Frame piramidale*: i microcluster sono conservati in una struttura che mantiene degli snapshot in diversi momenti. La struttura è di tipo piramidale, e gli snapshot sono conservati con livelli di granularità (e precisione) differenti a seconda che essi siano più o meno recenti. Il numero massimo di microcluster conservati è predefinito e costante: i microcluster meno recenti vengono aggregati per fare spazio ai nuovi microcluster generati leggendo il flusso in input.

DenStream

DenStream [8], è un algoritmo per il clustering basato su densità per flussi di dati evolventi. Anch'esso, come CluStream, utilizza la nozione di microcluster per mantenere una struttura sinottica dei punti del flusso precedentemente analizzati. Questa struttura, detta modello, è costituita da un insieme di micro-cluster.

DenStream esegue in due fasi: una prima fase online, nella quale si esegue l'aggiornamento del modello analizzando i nuovi punti in arrivo dal flusso dati, effettuando per ognuno di essi una ricerca esaustiva del più vicino micro-cluster presente nel modello, eventualmente aggregando il punto al micro-cluster o formandone uno nuovo; ed una seconda fase offline, nella quale si esegue il clustering dei micro-cluster del modello, trattandoli come fossero punti virtuali di un dataset statico, attraverso una variante della più nota tecnica DBScan [9].

L'algoritmo assume un modello a finestre smorzate (decaying windows), nel quale il peso di ogni tupla del flusso dati decresce esponenzialmente col tempo t secondo una funzione di fading esponenziale $f(t) = e^{-\lambda t}$ con $\lambda > 0$. Il peso totale del flusso è la costante $W = \frac{v}{1-2^{-\lambda}}$, dove v è la costante di velocità del flusso, ovvero il numero di punti che arrivano in elaborazione nell'unità di tempo. La funzione di fading fa sì che l'importanza dei dati storici decresca in funzione del tempo e all'aumentare del coefficiente dell'esponenziale λ.

In DenStream, il concetto di core point presente nel DBSCAN viene esteso e viene introdotta la nozione di micro-cluster, che conserva una rappresentazione approssimata di un gruppo di data point. In DBScan, un core point è un oggetto che ha almeno ε vicini, la cui somma dei pesi sia almeno pari a μ. Un clustering è un insieme di core point che hanno la stessa etichetta di cluster.

Si introducono tre nozioni di micro-cluster: i core-micro-cluster (c-micro-cluster), i potential core-micro-cluster (p-micro-cluster) e gli outlier micro cluster (o-micro-cluster).

Un *core-micro-cluster* al tempo t per un gruppo di punti vicini $p_{i_1}, \dots, p_{i_n}$ con timestamps $T_{i_1}, \dots, T_{i_n}$ è definito come $CMC(w, c, r)$ con w peso , c centro e r raggio del c-micro-cluster. Il peso $w = \sum_{j=1}^{n} f(t - T_{i_j})$ deve essere tale che $w \leq \mu$. Il centro è definito come:

$$c = \frac{\sum_{j=1}^{n} f\left(t - T_{i_j}\right) p_{i_j}}{w}$$

E il raggio come:

$$r = \frac{\sum_{j=1}^{n} f\left(t - T_{i_j}\right) dist(p_{i_j}, c)}{w}$$

Con $r \leq \varepsilon$. $dist(p_{i_j}, c)$ è la distanza euclidea tra il punto p_{i_j} e il centro c. Si noti che il peso di un micro-cluster deve essere maggiore di una soglia predefinita μ per poter essere considerato un core-micro-cluster. Un micro-cluster può comunque variare secondo i dati nel flusso, contentendone un calcolo incrementale.

Un c-micro-cluster potenziale (p-micro-cluster) al tempo t per un gruppo di punti tra loro vicini $p_{i_1}, \dots, p_{i_n}$ con timestamps $T_{i_1}, \dots, T_{i_n}$ è definito come $\{\overline{CF^1}, \overline{CF^2}, w\}$, dove il peso w deve essere tale che $w \geq \beta\mu$ con $0 < \beta \leq 1$. β è un parametro che definisce una soglia che distingua se il micro-cluster possa essere considerato o meno come outlier. $\overline{CF^1} = \sum_{j=1}^{n} f\left(t - T_{i_j}\right) p_{i_j}$ è la somma lineare dei pesi dei punti, mentre $\overline{CF^2} = \sum_{j=1}^{n} f\left(t - T_{i_j}\right) p_{i_j}^2$ è la somma pesata dei quadrati dei punti, inteso come quadrato del modulo del vettore posizione. Il centro di un p-micro-cluster è $c = \frac{\overline{CF_1}}{w}$ ed il suo raggio $r < \epsilon$ è $r = \sqrt{\frac{\overline{CF^2}}{w} - \left(\frac{\overline{CF^1}}{w}\right)^2}$.

Un p-micro-cluster è quindi un insieme di punti che potrebbe diventare un micro-cluster.

Un outlier micro-cluster (o-micro-cluster) al tempo t per un gruppo di punti tra loro vicini $p_{i_1}, \dots, p_{i_n}$ con timestamps $T_{i_1}, \dots, T_{i_n}$ è definito come $\{\overline{CF^1}, \overline{CF^2}, w, t_0\}$. Le definizioni di $\overline{CF^1}, \overline{CF^2}, w$, centro e raggio sono le stesse del p-micro-cluster, mentre $t_0 = T_{i_j}$ rappresenta l'istante di creazione dell'o-micro-cluster. In un o-micro-cluster il peso w deve essere inferiore a una soglia fissata, quindi $w < \beta\mu$. Può comunque crescere e diventare un p-micro-cluster se, aggiungendo nuovi punti, il pese supera la soglia.

L'algoritmo consta di due fasi: la fase online, nella quale i micro-cluster sono mantenuti e aggiornati a mano a mano che nuovi punti arrivano dal flusso; la fase offline, nella quale i cluster sono finalmente generati, a richiesta, dall'utente. Durante la fase online, quando un nuovo punto p è pronto, DenStream prova ad unirlo al più vicino micro-cluster c_p, soltanto se il raggio r_p di c_p non aumenta oltre ε, ovvero deve valere $r_p < \varepsilon$. Se questo vincolo non è soddisfatto, l'algoritmo tenta di aggregare il punto al più vicino o-micro-cluster c_o, a patto che $r_o < \varepsilon$. Si controlla quindi che il peso w deve sia tale che $w \geq \beta\mu$ ed in tal caso c_o viene promosso in un p-micro-cluster. Se non si verifica nessuna delle precedenti, un nuovo o-micro-cluster è generato dal punto p. Si noti che il peso di ogni p-micro-cluster esistente andrà a decadere gradualmente, secondo la funzione di fading, e, se non alimentato da nuovi punti, potrà essere declassata a o-micro-cluster, qualora il peso sia inferiore a $\beta\mu$.

La fase offline usa una variante dell'algoritmo DBSCAN, nella quale i p-micro-cluster sono considerati come punti virtuali. I concetti di densità-raggiungibile e densità-connesso sono usati da DenStream per generare il risultato finale.

D-Stream

L'algoritmo D-Stream [10], come il precedente, mira a computare in tempo reale dei cluster basati su densità. Come DenStream, anche D-stream utilizza un modello a finestre smorzate, secondo una del tutto simile funzione esponenziale, ma, a differenza di DenStream e CluStream, che utilizzano la nozione di micro-cluster, D-Stream basa la costruzione della struttura sinottica su delle griglie.

Concettualmente, D-Stream mantiene una struttura che discretizza tutte le dimensioni del dominio, e mantiene, per ogni cella della griglia, una stima del numero dei punti presenti in quella cella, cioè della densità in quella cella. Mantenendo questa struttura, è possibile eseguire un clustering basato su densità semplicemente aggregando le celle vicine con densità maggiore di una certa soglia ε.

In dettaglio, quindi, considerando che flusso in input sia composto da elementi di d dimensioni, ogni i-esima dimensione è discretizzata in p_i intervalli. In questo modo, sono definite un totale di $\eta = \prod_i p_i$ celle. Per ogni cella S è definito un *peso*, calcolato come la somma degli elementi appartenenti all'intervallo definito dalla cella, ognuno smorzato al tempo corrente con una funzione di decadimento del tutto simili a quella presentata per DenStream.

Ancora analogamente a DenStream, sono definite in [9] celle *dense*, *sparse* e *di transizione*, del tutto concettualmente simili ai c-microcluster, o-microcluster e p-microcluster. Una cella può cambiare stato in funzione del peso della cella al tempo corrente.

Anche D-stream esegue in due fasi, una online ed una offline: la prima fase mantiene ed aggiorna la griglia di celle man mano che nuovi elementi vengono letti dal flusso; la seconda fase calcola, a richiesta, i cluster considerando le sole celle *dense* della griglia, secondo una semplice regola: due celle sono connesse se sono *dense* e se sono adiacenti.

Il numero η di celle utilizzate dalla struttura dati, per come lo si è definito, è esponenzialmente dipendente dal numero di dimensioni d. Nella pratica, però, la maggior parte delle celle è completamente vuota, e le informazioni sulle celle vuote non hanno necessita di essere memorizzate. In [9] sono comunque definite delle tecniche per riconoscere e rimuovere le celle prodotte da un alto numero di outlier o dal rumore.

6. Bibliografia

1. Ravi Bhide, Flajolet-Martin algorithm, http://ravi-bhide.blogspot.it/2011/04/flajolet-martin-algorithm.html
2. Flajolet, P. and Martin, N., Journal of Computer and System Sciences, 1985. PDF - http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.81.3869&rep=rep1&type=pdf
3. Rajaraman, Anand, and Jeffrey David Ullman. Mining of massive datasets. Cambridge University Press, 2011.
4. Alon, Noga, Yossi Matias, and Mario Szegedy. "The space complexity of approximating the frequency moments." Proceedings of the twenty-eighth annual ACM symposium on Theory of computing. ACM, 1996.
5. Datar, Mayur, et al. "Maintaining stream statistics over sliding windows." SIAM Journal on Computing 31.6 (2002): 1794-1813.
6. S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering Data Streams. In IEEE FOCS Conference, 2000
7. L. O'Callaghan, N. Mishra, A. Meyerson, S. Guha, and R. Motwani. Streaming-Data Algorithms For High-Quality Clustering. In ICDE Conference, 2002.
8. F. Cao, M. Ester, W. Qian, A. Zhou. Density-based Clustering over an Evolving Data Stream with Noise, InSDM Conference, 2006.
9. M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In Proc of KDD, 1996.
10. Y. Chen, and L. Tu. Density-based clustering for real time stream data,ACM KDD Conference, 2007.
11. Rajaraman, Anand, and Jeffrey David Ullman. Mining of massive datasets. Cambridge University Press, 2011.
12. Bloom, Burton H. (1970), Space/Time Trade-offs in Hash Coding with Allowable Errors, Communications of the ACM 13 (7): 422–426.
13. C. Aggarwal. A Survey of Stream Clustering Algorithms, In "Data Clustering: Algorithms and Applications", ed. C. Aggarwal and C. Reddy, CRC Press, 2013.
14. Gini, Corrado, Measurement of Inequality and Incomes in The Economic Journal, vol. 31, 1921, pp. 124-126.
15. Muthukrishnan, S. Data streams: Algorithms and applications. Now Publishers Inc, 2005.