Università degli Studi di Padova
Dipartimento di Scienze Statistiche
Corso di Laurea Triennale in
Statistica per le Tecnologie e le Scienze

Relazione Finale

Stima di Frequenza in Stream di Dati tramite Predizioni Algoritmiche

Relatore Prof. Matteo Ceccarello
Dipartimento di Ingegneria dell'Informazione

Laureando Stefano Terrone
Matricola 2076734

Anno Accademico 2024/2025

Indice

1 Introduzione **4**

2 Related work **4**

3 Metodologia **5**

3.1 Algoritmo 6

4 Esperimenti **11**

4.0.1 Metriche di errore 12

4.1 Dataset Uniforme 12

4.1.1 Scelta della proporzione di divisione delle colonne della configurazione 13

4.1.2 Analisi dei risultati su dataset uniforme 15

4.2 Dataset Zipf 20

4.2.1 Scelta della proporzione di divisione delle colonne della configurazione 21

4.2.2 Analisi dei risultati su dataset Zipf 23

4.3 Dataset `CIC-DDoS2019` 28

4.3.1 Scelta della proporzione di divisione delle colonne della configurazione 29

4.3.2 Analisi dei risultati dataset `CIC-DDoS2019` 30

5 Conclusioni **35**

1 Introduzione

Questo lavoro rientra in un contesto di ricerca, detto *Algorithms with Predictions*, che si occupa di sviluppare framework che possano combinare algoritmi classici con dei predittori. Gli scopi e i guadagni di questo approccio possono essere molteplici, principalmente però si cerca di unire i vantaggi di entrambi i mondi. Mentre gli algoritmi classici permettono di avere delle garanzie di prestazione, spesso al caso peggiore, essi falliscono nello sfruttare eventuali pattern o informazioni a priori che si possono avere sul problema. Se da un lato il garantire delle prestazioni anche in casi di situazioni estreme è un vantaggio, il non considerare o sfruttare eventuali informazioni ottenute dai dati rappresenta sicuramente un'opportunità mancata. Proprio per questo motivo, l'affiancare un predittore che possa analizzare questi dati e guidare l'algoritmo, sembrerebbe essere una scelta molto logica.

2 Related work

Il mondo del machine learning e della statistica hanno sviluppato negli anni una grande varietà di modelli e tecniche che possono essere utilizzate per questo scopo. Come evidenziato da Mitzenmacher and Vassilvitskii (2020), l'approccio di questo metodo deve puntare all'avere degli algoritmi che possano garantire una performance ottimale quando il predittore fornisce la risposta corretta, ma che possano comunque garantire delle prestazioni accettabili anche quando il predittore sbaglia. Nonostante i guadagni possano essere molteplici, bisogna tenere conto di varie difficoltà che si possono presentare. Ad esempio bisogna fare attenzione a cosa predire, e allo stesso tempo capire quanto questo ci possa tornare utile. Un altro vantaggio di questo approccio è anche la sua potenziale generalizzabilità a tanti ambiti e tipi di problemi. Sia Lattanzi et al. (2020) che Sreenivas and Debmalya (2019) hanno usato questo approccio per sviluppare un algoritmo online per lo *Ski rental problem* (chiamato anche *rent-or-buy*), con l'obiettivo di usare le previsioni dell'oracolo per guidare la decisione. Oppure questo metodo è utilizzabile per il *Caching problem* come mostrato da Lykouris and Vassilvitskii (2020) i quali dimostrano

come seguire ciecamente le previsioni dell'oracolo possa portare a delle prestazioni scarse anche in presenza di un basso margine di errore. Invece, grazie ad una modifica del *Marker algorithm* tramite l'aggiunta di un predittore, che sia anche solo buono in media, si riesce a migliorare le prestazioni rispetto all'algoritmo *Least Recently Used (LRU)*, riducendo la *worst-case performance* del problema di caching. Kraska et al. (2018) presentano come un predittore possa essere usato per migliorare le prestazioni di varie strutture dati, in particolare si cerca di proporre un predittore che possa imparare l'ordine o la struttura dei dati, in modo da migliorare le prestazioni di una struttura dati classica come un *B-tree*. Altre applicazioni sono state provate come il *perfect hashing* e la *CDF Modelling*, *Bloom filter* e *learned Bloom filter* da Vaidya et al. (2020), la *page migration* da Indyk et al. (2020). L'applicazione di predittori a strutture dati classiche, come ad esempio ai Bloom filter si è rivelato essere un'idea molto prolifica e costituisce la base di questo lavoro.

3 Metodologia

Una delle applicazioni di questo approccio è la stima delle frequenze da un flusso di dati. Per la gestione di questo tipo di dati vengono usati degli algoritmi, detti *Streaming algorithms*, sviluppati con l'obbiettivo di lavorare con la minore quantità di memoria possibile. In uno dei primi studi su questo tipo di algoritmi, Alon et al. (1996) hanno dimostrato che i primi 3 momenti di una sequenza, come il numero di elementi unici, la lunghezza della sequenza e la frequenza di ripetizione degli elementi in un flusso di dati possano essere approssimati in uno spazio logaritmico. Gli *streaming algorithms* si dividono in due categorie principali: gli algoritmi *Counter Based* e gli *Sketch algorithms*. A quest'ultima classe appartiene la struttura dati *Count-Min Sketch* che viene usata in questo studio. Una *Count-Min Sketch*, abbreviata in *CM Sketch*, è una struttura dati probabilistica introdotta da Cormode and Muthukrishnan (2005). È una semplificazione della *Count Sketch* e della *AMS Sketch* può essere vista come una realizzazione di un *counting Bloom filter* o di un *Multistage-Filter*. Consiste in un array bidimensionale di dimensione d x w fissati a priori.

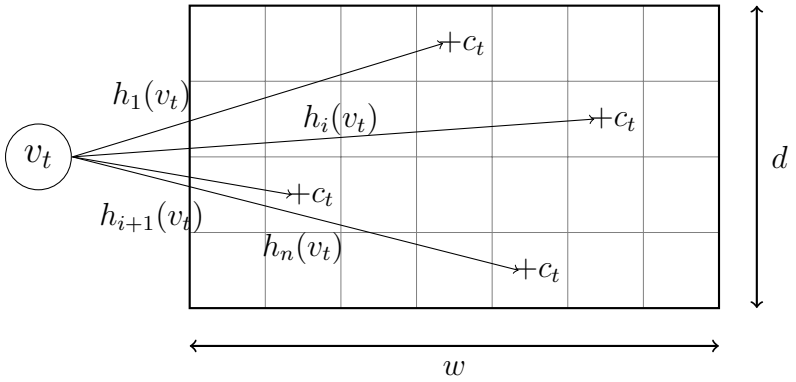

Figura 1: Funzionamento di una *Count-Min Sketch*.

La profondità d è il numero di funzioni hash usate per calcolare gli indici delle celle dell'array, mentre la larghezza w indica la lunghezza della hash table, ovvero il numero di bucket. Dunque, per ogni elemento del flusso di dati, viene calcolato un indice per ogni funzione hash e il valore dell'elemento viene sommato alla cella corrispondente nell'array. Per calcolare la frequenza di un elemento, si calcolano gli indici per ogni funzione hash e si restituisce il minimo valore tra le celle corrispondenti (Figura 1). L'algoritmo di questo studio si basa sul duplice uso della *Count-Min Sketch*, essa infatti può essere usata sia come filtro per eliminare i dati non frequenti, sia per calcolare le frequenze degli elementi. L'uso di questa struttura dati permette di ridurre notevolmente l'uso della memoria, considerato l'obbiettivo principale, cercando allo stesso tempo di mantenere una buona accuratezza e generalizzazione nella stima delle frequenze. L'approccio proposto in questo studio è non convenzionale rispetto a quelli classici, in quanto il predittore non è un modello di machine learning, ma una struttura dati probabilistica.

3.1 Algoritmo

L'algoritmo proposto in questo studio è un *streaming algorithm* che utilizza la *Count-Min Sketch* per calcolare le frequenze degli elementi in un flusso di dati. L'algoritmo è composto da due fasi principali: una fase di filtraggio e una fase di calcolo delle frequenze.

Algorithm 1 Funzione di Hash

```
Richiede: numero di righe rows
procedure HASH(element)
    element_bytes ← converti gli elementi in bytes
    indices ← inizializza lista vuota
    for i = 0 to rows do
        hash_value ← hash(element_bytes, seed = i)
        indices.append(abs(hash_value) % cols)
    end for
    return indices
end procedure
```

Il primo metodo permette di fare l'hash di un elemento dato in ingresso. Gli indici vengono calcolati per ogni funzione e sono uguali al numero di colonne della *Count Min Sketch*. L'implementazione di questo metodo è mostrata nell'Algoritmo 1. La funzione di hash può essere implementata in diversi modi, ma in questo studio si è scelto di usare la funzione di hash non crittografica *murmurHash*, in quanto non si ha necessità di sicurezza e risulta essere più veloce rispetto ad altre funzioni di hash crittografiche, come *SHA-256* o *MD5*. Per rendere unici gli indici di un singolo elemento nelle varie righe, la funzione di hash viene calcolata usando seed diversi per ogni riga della *Count Min Sketch*. Altre soluzioni potrebbero includere l'uso di un *salt crittografico* o l'uso di diverse funzioni di hash per ogni riga. Queste due soluzioni possono aiutare a ridurre le collisioni, al costo di aumentare il tempo di calcolo per le funzioni di hash e lo spazio in memoria.

Algorithm 2 Metodo Insert *Count-Min Sketch* doppia

```
Richiede: contatore esatto top K elementi counter, CMS per contare
  elementi frequenti count_matrix, insieme dei top K elementi q con la
  loro frequenza e timestamp
procedure INSERT(element, timer, counter=1)
    if elemento è in q then
        aggiorno il timestamp e la frequenza di elemento nel counter
        return
    end if
    if la coda non è ancora piena then
        inserisco l'elemento nel counter e aggiorno la frequenza
        return
    end if
    if la coda è piena ed elemento non è in q then
        query dell'elemento con il timestamp più vecchio
        if frequenza elemento vecchio minore frequenza elemento then
            sposto il vecchio elemento in count_matrix
            elimino il vecchio elemento dal counter
            aggiungo il nuovo elemento al counter con il timestamp corrente
        else
            incremento la frequenza dell'elemento corrente in count_matrix
        end if
    end if
end procedure
```

Il secondo metodo è quello che si occupa di calcolare le frequenze degli elementi che hanno passato il filtraggio della prima *Count-Min Sketch*. L'Algoritmo 2 mostra il funzionamento di questo metodo. Supponendo di avere un numero fissato di K elementi frequenti, possiamo utilizzare una struttura dati q che mantiene i K elementi con la loro frequenza e il timestamp. Affidandoci a q, possiamo gestire l'inserimento degli elementi in modo da contare in *counter* solo i K elementi più frequenti, in modo tale da consumare meno memoria. Una volta che un nuovo elemento diviene frequente, questo verrà inserito in q, spostando l'elemento con il timestamp più vecchio. La scelta ricade sull'elemento con timestamp più vecchio per mantenere una freschezza dei dati. Infatti avrebbe poco senso mantenere un elemento che non è più recente, in quanto potrebbe non ricomparire più e quindi occuperebbe

spazio inutilmente. L'elemento eliminato in *counter* non viene eliminato dalla *Count-Min Sketch*, ma viene spostato in *count_matrix*. Così che se dovesse tornare frequente, potrà essere recuperato e inserito nuovamente in *counter* senza dover ricalcolare la frequenza.

Algorithm 3 Metodo Filtro *Count-Min Sketch* doppia

```
    Richiede: numero di righe rows, numero di colonne cols, frequenza
      di aggiornamento count, CMS per filtrare filter_matrix, lista
3:    elementi frequenti presenti nel contatore esatto counter
    procedure FILTER(element)
        count ← incrementa di 1 il conteggio
6:      hash_indices ← calcola l'hash dell'element
        filter_matrix ← incrementa di 1 la frequenza nella prima CMS
        min_count(filter_matrix) ← trovo la frequenza dell'elemento
9:      if l'elemento supera la soglia, o se elemento già presente in counter
    then
            insert(element) ← inserisco l'elemento, o ne aggiorno la frequenza
    se è già presente
        end if
12: end procedure
```

L'Algoritmo 3 mostra il Funzionamento del filtro composto dalla prima *Count-Min Sketch*. Il suo scopo è filtrare gli elementi non frequenti da quelli frequenti, facendo poi passare questi ultimi alla seconda *Count-Min Sketch*, o al contatore esatto che si occuperà di contarli in modo preciso. L'algoritmo inizia incrementando il conteggio della frequenza che fungerà da timestamp, e dopo aver calcolato l'hash dell'elemento usando il metodo spiegato nell'Algoritmo 1, incrementa la frequenza nella prima *Count-Min Sketch*. Se l'elemento si trova già tra gli elementi frequenti l'algoritmo aggiorna la frequenza e il timestamp nel dizionario tramite il metodo *insert* spiegato nell'Algoritmo 2. Se invece l'elemento non è presente, viene verificato se soddisfa la condizione per diventare un elemento frequente. Nel caso dovesse farlo, l'elemento verrà spostato nella seconda *Count-Min Sketch* con il timestamp aggiornato e frequenza pari a quella della prima *Count-Min Sketch*, altrimenti verrà ignorato. Nella implementazione trattata nello studio, per risparmiare spazio, non viene tenuta traccia degli elementi che hanno superato la soglia, ma per capire

quali elementi sono frequenti, viene confrontata la frequenza dell'elemento corrente con la frequenza del più vecchio tra gli elementi frequenti. Un'altra tecnica utilizzabile, particolarmente utile nel caso di volesse avere una soglia dinamica, è l'aggiornamento usando una procedura *EWMA (Exponential Weighted Moving Average)*. Nonostante questa procedura non sia tipica per questo tipo di algoritmi, alcuni studi, come da Čisar and Čisar (2007), ne hanno mostrato la potenziale utilità nel creare soglie adattive. La *Tecnica di Controllo EWMA* dipende direttamente dalla *Statistica EWMA*. Questa statistica, formulata da Roberts (2000), è utilizzata per monitorare e controllare i processi nel tempo, adattando dinamicamente la soglia in base ai dati storici. La procedura EWMA aggiorna la soglia $EWMA_t$ secondo la formula:

$$EWMA_t = \alpha \cdot Y_t + (1 - \alpha) \cdot EWMA_{t-1} \tag{1}$$

dove Y_t è la frequenza osservata al tempo t, $EWMA_{t-1}$ è la soglia precedente e $\alpha \in (0, 1)$ è il parametro di smoothing che determina quanto peso dare alle osservazioni più recenti rispetto a quelle passate. Un valore di α vicino a 1 rende la soglia dipendente maggiormente dalle osservazioni recenti, mentre un valore più basso da più peso alle osservazioni passate. Questa soluzione è usata da Zang et al. (2025) per creare una soglia dinamica in un algoritmo che combina due strutture dati, una *Direct Bitmap* e una *Count-Min Sketch*.

Algorithm 4 Metodo query *Count-Min Sketch* doppia

```
Richiede: CMS per contare elementi frequenti count_matrix,
  contatore esatto counter
procedure QUERY(element)
    hash_indices ← hash(element)
    if elemento è in counter then
        freq_counter ← counter[element]
    else
        freq_counter ← 0
    end if
    min_count(count_matrix) ← trovo la frequenza dell'elemento
     return min_count + freq_counter
end procedure
```

L'ultimo metodo serve a recuperare la stima della frequenza di un elemento frequente. Se questo elemento si trova nel contatore esatto, la sua frequenza è la somma della frequenza calcolata dalla *Count-Min Sketch* e quella del contatore esatto. Questo perchè quando diventa un elemento frequente, e viene spostato dalla *Count-Min Sketch* al contatore esatto, la sua frequenza viene mantenuta nella *Count-Min Sketch*, senza però venire aggiornata, finchè l'elemento torna ad essere non frequente.

4 Esperimenti

Al fine di valutare le prestazioni dell'algoritmo proposto, sono stati condotti diversi esperimenti su dataset reali e sintetici. I dataset valutati includono:

- un dataset sintetico composto da sequenze di eventi generati casualmente da una distribuzione uniforme

- un dataset sintetico composto da sequenze di eventi generati casualmente da una distribuzione Zipf

- un dataset reale contenente dati di traffico di rete, forniti dall'università del New Brunswick *(https://www.unb.ca/cic/datasets/ddos-2019.html)*.

I due dataset sintetici hanno numerosità 1 milione di dati ciascuno. Le simulazioni sono state eseguite su due *Count-Min Sketch*. La prima *Count-Min Sketch* è una classica *Count-Min Sketch*, a cui è stato aggiunto un dizionario che permette di contare gli *Heavy Hitters*. Funge da confronto con la *Count-Min Sketch* doppia. La *Count-Min Sketch* doppia è stata implementata seguendo l'approccio descritto nel capitolo *3.1 algoritmo*. L'algoritmo proposto è stato scritto in `Python` *(https://www.python.org/)* versione 3.13, il codice è stato compilato usando la libraria `Cython` *(https://cython.org/)* per migliorare le prestazioni. Le simulazioni sono state eseguite su un computer con processore `Intel Core i3-1315U`, 8 GB di RAM e sistema operativo `Windows 11 versione 24H2`.

4.0.1 Metriche di errore

Le metriche di errore considerate per la valutazione delle prestazioni includono l'errore relativo medio, il throughput medio e la sovrastima media. L'errore relativo medio è definito come:

$$\frac{|f - \hat{f}|}{f} \tag{2}$$

Dove f è la frequenza reale dell'elemento e $\hat{f}$ è la frequenza stimata. Il throughput medio è definito come il numero di elementi elaborati per unità di tempo. La sovrastima media è definita come:

$$\frac{1}{N}\sum_{i=1}^{N}(\hat{f} - f) \tag{3}$$

Dove N è il numero totale di elementi nel flusso. Corrisponde alla media della differenza tra la frequenza stimata degli *Heavy Hitters* e la loro frequenza reale.

4.1 Dataset Uniforme

Il seguente dataset è stato generato casualmente utilizzando due distribuzioni uniformi. La sequenza di dati generati ha una lunghezza di 100.000 elementi. Il dataset è composto in modo da avere tanti dati frequenti all'inizio concentrati in poche posizioni, seguiti da un calo netto di frequenza. Questo dataset è stato scelto come caso limite per testare l'algoritmo proposto, in quanto simula un flusso di dati con un elevato picco di frequenza per pochi elementi, con i restanti che presentano frequenze molto basse.

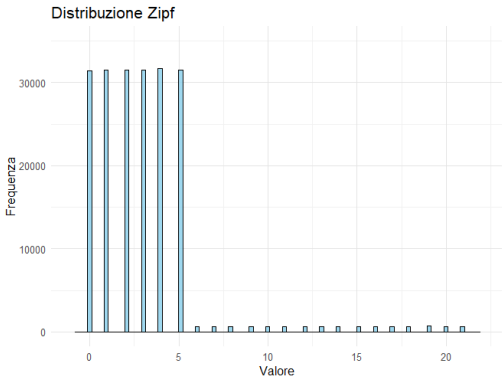

Figura 2: Istogramma dei primi 20 valori della distribuzione uniforme in analisi.

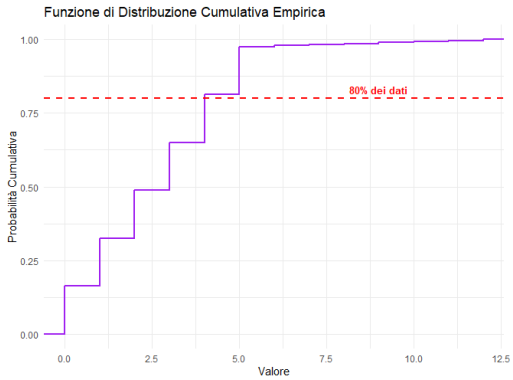

Figura 3: Funzione di distribuzione cumulativa dei primi 12 valori della distribuzione uniforme in analisi.

Come mostrato in Figura 2, l'istogramma evidenzia come gli elementi frequenti siano concentrati nelle prime 6 posizione, e successivamente vi è una rapida diminuzione della frequenza degli elementi. Dalla Figura 3 si può osservare come solo i primi 5 elementi rappresentino quasi l'80% dei dati e, se si dovessero considerare tutti e 6 gli elementi frequenti, essi rappresentano il 90% di tutti i dati nel flusso.

4.1.1 Scelta della proporzione di divisione delle colonne della configurazione

Si procede ora a verificare, per varie configurazioni di colonne, quale proporzione di divisione meglio si adatta a questo dataset. Per la verifica è stato usato un dataset uniforme, contenente 100.000 elementi e 6 *Heavy Hitters*, sono state scelte tre diverse configurazioni di colonne: una leggera da 30 colonne che utilizza circa $1Kb$ di memoria, una configurazione media da 110 colonne che utilizza circa $2Kb$ e una configurazione pesante da 350 colonne, di circa $4.7Kb$.

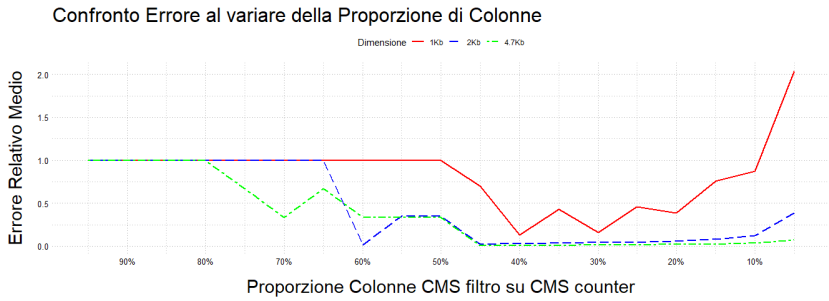

Figura 4: Evoluzione dell'errore in funzione della proporzione di divisione delle colonne su dataset uniforme.

Dalla Figura 4 si osserva come nella prima configurazione da $1Kb$, l'errore medio presenti valori molto alti di errore rispetto alle altre due configurazioni. Il valore più basso si osserva nella configurazione in cui la *Count-Min Sketch* filtro ha un proporzione di 1 a 4 e di 1 a 3 sulla *Count-Min Sketch* conteggio. Per la configurazione da $2Kb$, l'errore medio minore si osserva in corrispondenza della proporzione di divisione del *60%*, comunque questa configurazione mantiene valori bassi di errore per tutte le proporzioni che vanno dal 45% al 10%. La configurazione da $5Kb$ mantiene un errore basso per tutte le proporzioni a partire dalla proporzione del *45%*, superata la soglia del *20%*, come per la configurazione da $2Kb$, l'errore comincia ad aumentare. Nella prima parte, tutte le configurazioni presentano un errore medio pari ad 1. Questo perchè il valore 1 è legato alla mancata identificazione di tutti gli *Heavy Hitters*. Dunque, le configurazioni con un numero di colonne dedicate al filtraggio troppo alto, non riescono ad identificare tutti gli *Heavy Hitters*, e quindi presentano un errore medio molto alto. Inoltre si può notare come più le configurazioni sono grandi, questo problema diventi meno evidente, risolvendosi anticipatamente. In generale, si può osservare come l'errore medio sembri diminuire all'aumentare della proporzione di colonne dedicate al conteggio. Sembrano dunque avvantaggiate le configurazioni con un maggior numero di colonne dedicate al conteggio, evidenziando l'importanza di questa fase nell'algoritmo proposto.

4.1.2 Analisi dei risultati su dataset uniforme

Si passa ora a verificare le prestazioni della *Count-Min Sketch* doppia su questo dataset. Dai risultati ottenuti nella sezione precedente, si è deciso di utilizzare una divisione al 40% delle colonne per il filtraggio e al 60% per il conteggio. Sono state create diverse configurazioni della *Count-Min Sketch* doppia, con un numero di colonne che varia da $0.90Kb$ a $5.5Kb$, e sono state eseguite 100 simulazioni per ogni configurazione. La stessa configurazione è stata provata anche su una *Count-Min Sketch* classica, per poter confrontare i risultati.

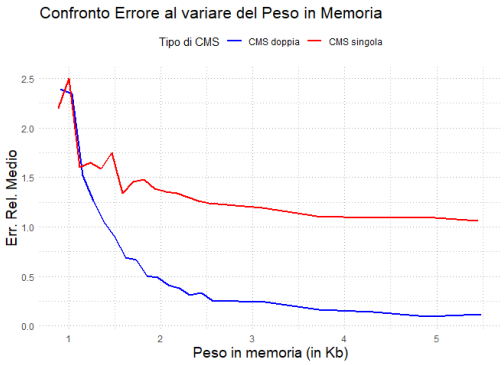

Figura 5: Evoluzione dell'errore in funzione della memoria in *KiloByte* su dataset uniforme.

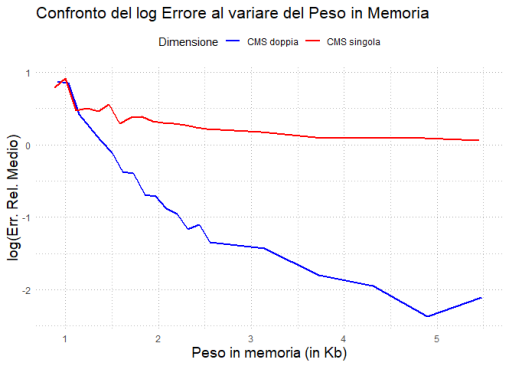

Figura 6: Evoluzione dell'errore su scala logaritmica in funzione della memoria in *KiloByte*.

Dalla Figura 5 si può osservare come l'errore medio diminuisca all'aumentare della memoria occupata dalla struttura dati. La *Count-Min Sketch* semplice presenta un errore medio maggiore rispetto alla *Count-Min Sketch* doppia. In alcuni punti la *Count-Min Sketch* doppia risulta essere uguale in termini di errore alla *Count-Min Sketch* semplice, come si può osservare superato $1Kb$ di memoria. L'errore parte molto elevato per entrambe le strutture, con un errore medio che oscilla tra 2.5 e 1.0 per le configurazioni più piccole. A differenza della *Count-Min Sketch* semplice, l'errore della *Count-Min Sketch* doppia continua a diminuire con l'avanzare della memoria. La *Count-Min Sketch* singola infatti non scende mai sotto l'errore di 1, suggerendo che in tutte le configurazioni gli elementi vengano contati con frequenza pari al doppio della loro frequenza reale, o che alcuni elementi

vengano completamente trascurati. La Figura 6 mostra lo stesso grafico della Figura 5, ma con scala logaritmica sull'asse delle ordinate.

Accuratezza nelle stime

Una prima analisi della precisione della struttura dati in analisi può essere effettuata confontando il gruppo degli *Heavy Hitters* individuati dalla *Count-Min Sketch* con il gruppo di *Heavy Hitters* reali.

Memoria (KB)	< 1.25	1.25-2.25	2.26-3.25	3.26-4.25	> 4.25
N. di HH	3.58	5.92	5.99	5.88	6.0
Precisione	0.596	0.986	0.999	0.980	1.0

Tabella 1: Numero di *Heavy Hitters* correttamente identificati e precisione (in percentuale) della *Count-Min Sketch* doppia.

Dalla Tabella 1 si può osservare come la *Count-Min Sketch* doppia riesca ad identificare correttamente quasi sempre tutti e 6 gli *Heavy Hitters* a partire dalla fascia dai $1.25Kb$ di memoria. La precisione per la *Count-Min Sketch* doppia con memoria inferiore all'$1.25Kb$ è circa del 60%, quindi riesce ad identificare mediamente tra i 3 e i 4 *Heavy Hitters* su 6. Per le configurazioni da $1.25Kb$ a $2.25Kb$ si ha una precisione del 97%, il che significa che queste configurazioni riescono ad identificare correttamente quasi sempre i 6 *Heavy Hitters* reali. Per le *Count-Min Sketch* doppie con memoria da $2.26Kb$ a $3.25Kb$ si ha una precisione del 99%, mentre per quelle da $3.26Kb$ a $4.25Kb$ la precisione diminuisce leggermente al 98%. Dunque, queste configurazioni riescono ad identificare quasi sempre tutti e 6 gli *Heavy Hitters* reali. Infine, le configurazioni con memoria superiore a $4.25Kb$ riescono ad identificare correttamente tutti gli *Heavy Hitters*, infatti come si può notare dalla Figura 5 l'errore è quasi pari a 0.

Metriche di Errore

Dopo aver analizzato l'accuratezza delle stime della *Count-Min Sketch* doppia, si passa ad analizzare alcune metriche di errore.

CMS semplice	Dimensione	N. Col	Errore	Throughput	Sovrastima
< 1.25	1.06	25	1.98	1.05 $\times 10^6$	70587
1.25-2.25	1.41	55	1.66	1.19 $\times 10^6$	42034
2.26-3.25	1.64	75	1.39	1.16 $\times 10^6$	28294
3.26-4.25	1.87	95	1.43	1.22 $\times 10^6$	39881
> 4.25	3.28	215	1.21	1.18 $\times 10^6$	35919

Tabella 2: Tabella di sintesi della *CMS* semplice di profondità 3 al variare della larghezza. Sono riportati i valori medi di dimensione (in *Kb*), larghezza media (in colonne), errore medio, throughput medio e sovrastima media.

Osservando la Tabella 2 si possono notare alcune statistiche delle varie configurazioni della *Count-Min Sketch* semplice. Le configurazioni di *Count-Min Sketch* semplice che occupano meno di 1.25*Kb*, presentano una larghezza media di 25 colonne e hanno un errore medio di 1.98, e una sovrastima media di 70587 elementi. La configurazione più piccola presenta un errore medio molto alto, e anche la sua sovrastima media risulta essere la più alta tra tutte le configurazioni. Questo e' dovuto al fatto che questa configurazione ha un precisione del 25%, dunque riesce ad identificare in media solo tra gli 1 e i 2*Heavy Hitters*, e quindi la sua sovrastima risulta essere gonfiata da elementi sovrastimati che sono stati considerati erroneamente degli *Heavy Hitters*. Le configurazioni che occupano tra 1.25*Kb* e 2.25*Kb* presentano un errore medio pari a 1.66, e una sovrastima media di 42034 elementi, risultando essere nettamente migliori delle configurazioni più piccole. Le configurazioni da 2.26*Kb* a 3.25*Kb* e da 3.26*Kb* a 4.25*Kb* presentano un errore medio rispettivamente di 1.39 e 1.43, con sovrastime medie di 28294 e 39881 elementi, con la prima che rappresenta il valore di sovrastima più basso per le configurazioni della *Count-Min Sketch* semplice. Le configurazioni che occupano più di 4.25*Kb* presentano un errore medio di 1.21 e una sovrastima media di 35919 elementi, di poco inferiore rispetto alla configurazione precedente.

CMS doppia	Dimensione	N. Col	Errore	Throughput	Sovrastima
< 1.25	1.03	20	2.01	7.81 $\times 10^5$	40259
1.25-2.25	1.73	80	0.71	8.19 $\times 10^5$	21527
2.26-3.25	2.61	155	0.29	8.26 $\times 10^5$	8995
3.26-4.25	3.73	250	0.17	8.13 $\times 10^5$	3997
> 4.25	4.89	350	0.13	8.14 $\times 10^5$	3756

Tabella 3: Tabella di sintesi della *CMS* doppia di profondità 3 al variare della larghezza. Sono riportati i valori medi di dimensione (in *Kb*), larghezza media (in colonne), errore medio, throughput medio e sovrastima media.

Confrontando la Tabella 2 con la Tabella 3, contenente alcune statistiche della *Count-Max Sketch*, si può osservare come la *Count-Min Sketch* doppia abbia un errore medio migliore rispetto alla *Count-Min Sketch* semplice, come già evidenziato nella Figura 4.

Sappiamo inoltre dalla Tabella 1 che questa configurazione riesce ad identificare in media 3.58 *Heavy Hitters*, contro gli 1.5 della *Count-Min Sketch* semplice, questo può significare che la configurazione più piccola della *Count-Min Sketch* non sovracconta gli elementi erroneamente classicati come *Heavy Hitters*, alla stessa maniera della *Count-Min Sketch* semplice. Aumentando la dimensione della struttura, si osserva netto un miglioramento delle prestazioni: la configurazione da $1.25Kb$ a $2.25Kb$ mostra un errore medio che scende da 2.01 a 0.71 e una sovrastima media pari a circa la metà delle configurazioni precedenti. Le configurazioni tra $2.26Kb$ e $3.25Kb$ mostrano un errore medio di 0.29 e una sovrastima media di 8995 elementi, un risultato migliore addirittura della sovrastima delle configurazioni più grandi della *Count-Min Sketch* semplice, che pesano più di $1Kb$ di memoria in più. Passando alle configurazioni più grandi, la *Count-Min Sketch* doppia raggiunge un errore medio quasi pari a 0.13, con una sovrastima inferiore a 4000 elementi già per configurazioni che occupano mediamente meno di $4.25Kb$. Le configurazioni più grandi mostrano risultati di poco superiori a quest'ultima configurazione. In termini di throughput medio, la *Count-Min Sketch* doppia risulta essere peggiore alla controparte semplice, penalizzata

soprattutto dal maggiore numero di operazioni necessarie per l'inserimento e la stima degli elementi.

Analisi dei tempi di esecuzione

Una ultima metrica da considerare è il tempo di esecuzione delle due strutture.

CMS 1	Min.	1° qt.	Mediana(IQR)	Media(sd.)	3° qt.	Max.
< 1.25	0.773	0.887	0.940(0.10)	0.950(0.12)	0.985	2.029
1.25-2.25	0.713	0.793	0.834(0.08)	0.846(0.08)	0.872	1.177
2.26-3.25	0.721	0.801	0.855(0.11)	0.864(0.09)	0.914	1.132
3.26-4.25	0.715	0.782	0.805(0.06)	0.822(0.07)	0.840	1.241
> 4.25	0.714	0.787	0.828(0.09)	0.846(0.09)	0.882	1.714
CMS 2	**Min.**	**1° qt.**	**Mediana(IQR)**	**Media(sd.)**	**3° qt.**	**Max.**
< 1.25	1.122	1.217	1.265(0.099)	1.285(0.14)	1.316	2.700
1.25-2.25	1.071	1.166	1.197(0.090)	1.222(0.09)	1.256	2.145
2.26-3.25	1.075	1.162	1.189(0.081)	1.214(0.10)	1.243	1.747
3.26-4.25	1.096	1.162	1.202(0.099)	1.227(0.10)	1.261	1.532
> 4.25	1.095	1.160	1.201(0.094)	1.228(0.11)	1.254	1.940

Tabella 4: Tabella di sintesi del tempo di esecuzione in secondi di una *Count-Min Sketch* semplice (`CMS 1`) e di una *Count-Min Sketch* doppia (`CMS 2`) di profondità 3 al variare della larghezza.

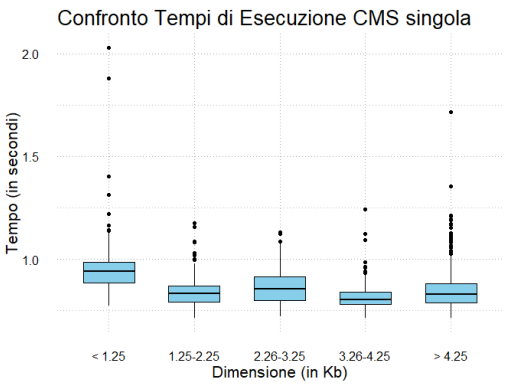

Figura 7: Boxplot dei tempi di esecuzione delle *Count-Min Sketch* singola.

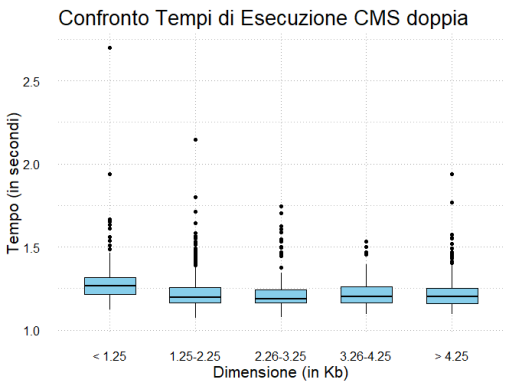

Figura 8: Boxplot dei tempi di esecuzione delle *Count-Min Sketch* doppia.

Come si può osservare dalla Tabella 4 la *Count-Min Sketch* doppia risulta essere più lenta della *Count-Min Sketch* semplice in tutte le configurazioni. Le distribuzioni dei tempi di esecuzione, nonostante non presentino valori di *deviazione standard* elevati, risultano essere asimmetriche, in particolare sono presenti elevati valori anomali, come osservato in Figura 7 e Figura 8. Le configurazioni più veloci della *Count-Min Sketch* singola risultano essere quelle che occupano che i $3.26Kb$ e i $4.25Kb$, con un tempo di esecuzione medio di $0.822s$ e un tempo mediano di $0.805s$. Per la *Count-Min Sketch* doppia le configurazioni più veloci risultano essere in media quelle che occupano i $2.26Kb$ e i $3.25Kb$, con un tempo di esecuzione medio di $1.214s$ e un tempo mediano di $1.189s$.

4.2 Dataset Zipf

Il secondo dataset analizzato è caratterizzato da una distribuzione di Zipf. Questo dataset è composto in modo da avere pochi elementi molto frequenti all'inizio, seguiti da elementi sempre meno frequenti. Questa distribuzione, che prende il nome dal matematico *George Zipf*, è stata prima utilizzata nel campo della semantica, ma le sue applicazioni si estendono a molti altri campi. Uno di questi campi è la modellazione del traffico Web. La *legge di Zipf* afferma che la frequenza di un elemento è inversamente proporzionale al suo rango. Dunque pochi elementi rappresentano la maggior parte del traffico, mentre molti altri elementi hanno frequenze molto basse. Come mostrato da Adamic and Huberman (2001) e da Breslau et al. (1999), la distribuzione di Zipf può essere utilizzata per descrivere problemi di *Web Caching* e di *Network* Oltre che alla ricerca via web, è stato dimostrato come la distribuzione di Zipf possa descrivere accuratamente la distribuzione degli indirizzi IP. Ad esempio, Bader (2022) dimostra come una distribuzione di Zipf-Mandelbrot sia in grado di descrivere in modo accurato un'ampia varietà di statistiche di origine/destinazione su campioni in movimento di indirizzi IP che vanno da 100.000 a 100.000.000 elementi. Il seguente dataset è stato generato casualmente utilizzando una distribuzione Zipf con parametro $s = 1.7$. La sequenza di elementi generati ha una lunghezza di 1 milione di eventi.

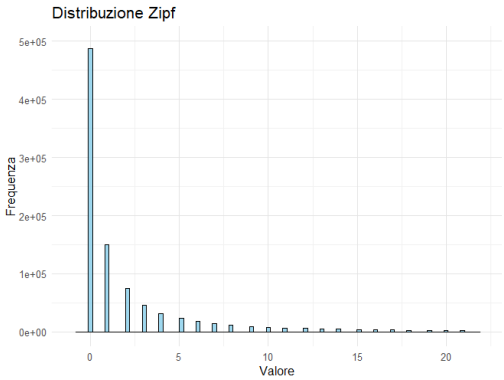

Figura 9: Istogramma dei primi 20 valori della distribuzione Zipf in analisi.

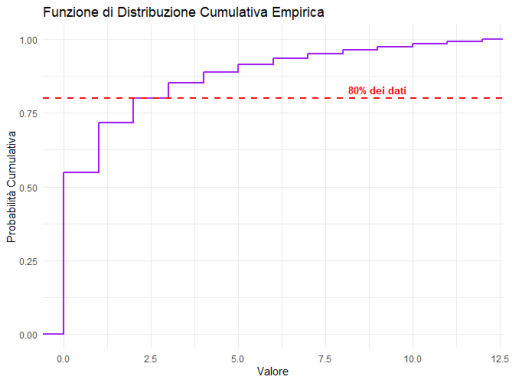

Figura 10: Funzione di distribuzione cumulativa dei primi 12 valori della distribuzione Zipf in analisi.

In Figura 9 è mostrato l'istogramma della distribuzione Zipf, si può osservare che la maggior parte dei dati è concentrata nelle prime posizioni, e che la frequenza diminuisce rapidamente. In Figura 10 è mostrata la funzione di distribuzione cumulata. Si può osservare come i primi 3 valori rappresentino quasi l'80% dei dati.

4.2.1 Scelta della proporzione di divisione delle colonne della configurazione

Si procede ora a verificare, per varie configurazioni di colonne della struttura dati, quale proporzione di divisione meglio si adatta a questo dataset. Per la verifica è stato usato un dataset contenente 1 milione di dati distribuiti secondo una distribuzione Zipf, e sono state usate le stesse configurazioni di dimensione analaga alla sezione precedente: una configurazione da 40 colonne, una da 130 colonne e una da 370 colonne, di dimensioni rispettivamente pari a $1Kb$, $2Kb$ e $5Kb$.

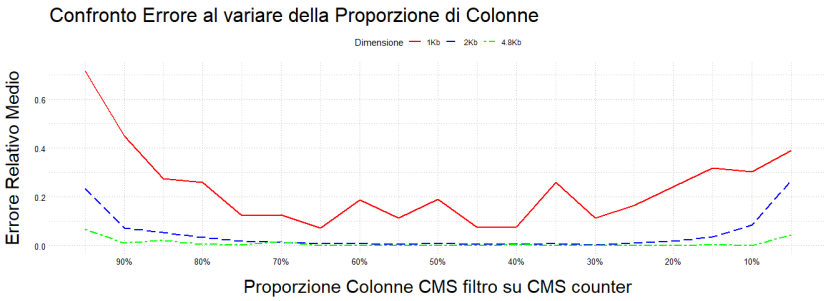

Figura 11: Evoluzione dell'errore in funzione della proporzione di divisione delle colonne su dataset Zipf.

Dalla Figura 11 si può osservare come l'errore medio evolve al diminuire della proporzione di colonne. In generale, si nota come nella configurazione da $1Kb$, l'errore medio appaia maggiore rispetto alle altre due configurazioni. In questa configurazione, l'errore medio minore lo si osserva in corrispondenza delle proporzioni di divisione del 65% e anche del 45% e 40%, dunque le divisioni migliori per questa configurazione sembrano essere quelle che rappresentano un equilibrio tra le dimensioni delle colonne della la *Count-Min Sketch* filtro e della *Count-Min Sketch* counter. I picchi di errore osservati in questa configurazione si osservano alle estremità, in prossimità di configurazioni con dimensioni delle due matrici estreme, e sono dovuti alla mancata identificazione di alcuni *Heavy Hitters*. In particolare, tutte le proporzioni con errore superiore a 0.25, riescono mediamente ad identificare solo 2 degli 3 *Heavy Hitters* reali. Nella configurazione da $2Kb$, l'errore medio assume valori molto bassi fino dal 65% al 30% di proporzione, successivamente si osservano alcuni picchi che portano l'errore medio a valori più elevati. In questa configurazione sono presenti solo due picchi di errore superiori al 0.20, in corrispondenza del 95% e del 5% di proporzione. Infine, per la configurazione da $5Kb$, l'errore medio risulta essere molto basso per tutte le proporzioni e sembra seguire lo stesso andamento della configurazione da $1Kb$ e da $2Kb$. Per entrambe le configurazioni da $2Kb$ e da $5Kb$, il valore di errore medio più basso lo si osserva in corrispondenza del 40% e del 30% di proporzione di divisione. Ad esempio, per la configurazione a $2Kb$, questo equivale a dire che la *Count-Min Sketch* filtro è composta rispettivamente da 52 o da 39

colonne, e la *Count-Min Sketch* counter è composta da 78 o da 91 colonne. In quanto l'errore medio più basso per entrambe le configurazioni si osserva in corrispondenza del 40% e del 30% di proporzione di divisione, ed essendo la prima anche la scelta migliore per la *Count-Min Sketch* a da $1Kb$, si è deciso di procedere utilizzando questa proporzione per le successive analisi.

4.2.2 Analisi dei risultati su dataset Zipf

Sono state eseguite delle simulazioni sul dataset di dati Zipf in analisi. In particolare, sono state effettutate 100 simulazioni su diverse configurazioni della *Count-Min Sketch*. La configurazione più piccola pesa $0.7Kb$, quella più grande pesa $5.1Kb$. Per confronto, le stesse configurazioni sono state provate su una *Count-Min Sketch* classica. Di seguito sono riportati i risultati:

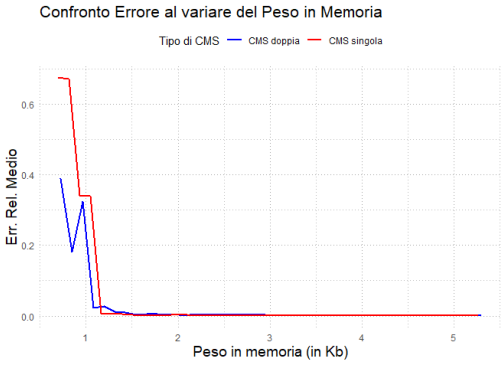

Figura 12: Evoluzione dell'errore in funzione della memoria in *KiloByte* su dataset Zipf.

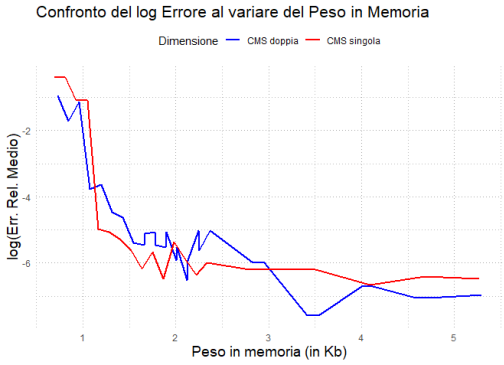

Figura 13: Evoluzione dell'errore su scala logaritmica in funzione della memoria in *KiloByte*.

Dalla Figura 12 si può osservare come l'errore medio diminuisca all'aumentare della memoria occupata dalla struttura dati. La *Count-Min Sketch* doppia presenta un andamento molto simile alla *Count-Min Sketch* semplice. Dopo questi picchi, l'errore medio della *Count-Min Sketch* doppia risulta quasi uguale a quello della *Count-Min Sketch* semplice, e diventa prossimo allo zero. In Figura 13 è mostrato lo stesso grafico della Figura 12, ma con scala logaritmica sull'asse delle ordinate. Si può osservare meglio l'evoluzione dell'errore medio. La *Count-Min Sketch* parte con un errore medio inferiore,

ma appena supera $1Kb$ di memoria, l'errore medio della *Count-Min Sketch* singola diventa inferiore a quest'ultimo e lo rimane fino ai $3Kb$, tranne per una eccezione intorno al valore di $2Kb$.

Accuratezza nelle stime

Alla luce dei risultati della sezione precedente, si è deciso di procedere utilizzando la proporzione di divisione delle colonne al 70%. Si è dunque passati a verificare la precisione della struttura dati nell'identificazione degli *Heavy Hitters*al variare della larghezza. Gli *Heavy Hitters* reali sono 3, e sono gli elementi che compaiono più frequentemente nella sequenza di dati.

Memoria (KB)	< 1.25	1.25-2.25	2.26-3.25	3.26-4.26	> 4.26
N. di HH	2.55	2.97	3.0	3.00	3.00
Precisione	0.85	0.99	1.00	1.00	1.00

Tabella 5: Numero di *Heavy Hitters* correttamente identificati e precisione (in percentuale) della *Count-Min Sketch* doppia.

I risultati mostrati in Tabella 5 mostrano quanti *Heavy Hitters* la *Count-Min Sketch* doppia riesca ad identificare correttamente all'aumentare dalla memoria occupata. In particolare, le configurazioni con memoria inferiore a $1.25Kb$ hanno una precisione dell'85%, il che significa che riesce ad identificare correttamente tutti e 3 gli *Heavy Hitters* 5 volte su 6. Per le configurazioni con memoria compresa tra $1.25Kb$ e $2.25Kb$, la precisione raggiunge il 99%, e successivamente, le configurazioni con memoria superiore a $1.75Kb$ mostrano una precisione del 100%.

Metriche di Errore

Dopo aver analizzato la capacità della *Count-Min Sketch* doppia di identificare gli *Heavy Hitters*, si passa ad analizzare alcune metriche di errore.

CMS semplice	Dimensione	N. Col	Errore	Throughput	Sovrastima
< 1.25	0.96	32	0.37	2.00 $\times 10^6$	3939
1.25-2.25	1.36	67	0.027	2.24 $\times 10^6$	811
2.26-3.25	1.63	90	0.0034	2.25 $\times 10^6$	508
3.26-4.26	1.93	115	0.0031	2.24 $\times 10^6$	482
> 4.26	3.39	240	0.0019	2.21 $\times 10^6$	253

Tabella 6: Tabella di sintesi della *CMS* semplice di profondità 3 al variare della larghezza. Sono riportati i valori medi di dimensione (in Kb), larghezza media (in colonne), errore medio, throughput medio e sovrastima media.

Osservando la Tabella 6 si possono notare alcune statistiche delle varie configurazioni della *Count-Min Sketch* semplice. Le configurazioni che occupano meno di $1.25Kb$ hanno una dimensione media di $0.96Kb$, un errore medio di 0.37 e una sovrastima media di 3939 elementi. Le configurazioni che occupano tra 1.25 e $2.25Kb$ hanno una dimensione media di $1.36Kb$, un errore medio di 0.027 e una sovrastima media di 811 elementi, di molto inferiori rispetto alla configurazione precedente. Le configurazioni che occupano tra 2.26 e $3.25Kb$ hanno una dimensione media di $1.63Kb$, un errore medio di 0.0034 e una sovrastima media di 508 elementi. Le configurazioni da 3.26 a $4.26Kb$ hanno una dimensione media di $1.93Kb$, un errore medio di 0.0031 e una sovrastima media di 482 elementi. Infine, per le strutture che occupano più di $4.26Kb$ si osserva una dimensione media di $3.39Kb$, un errore medio di 0.0019 e una sovrastima media di 253 elementi. Il throughput medio è molto simile per tutte le configurazioni, con un valore medio di circa 2.2×10^6 elementi al secondo, tranne per le configurazioni con memoria inferiore a $1.25Kb$ che hanno il throughput medio peggiore, pari a 2.00 $\times 10^6$ elementi al secondo.

CMS doppia	Dimensione	N. Col	Errore	Throughput	Sovrastima
< 1.25	0.97	31	0.18	1.30 $\times 10^6$	1374
1.25-2.25	1.75	99	0.0065	1.30 $\times 10^6$	745
2.26-3.25	2.71	182	0.0040	1.31 $\times 10^6$	393
3.26-4.25	3.77	275	0.0009	1.32 $\times 10^6$	154
> 4.25	4.95	375	0.0009	1.33 $\times 10^6$	109

Tabella 7: Tabella di sintesi della *CMS* doppia di profondità 3 al variare della larghezza. Sono riportati i valori medi di dimensione (in Kb), larghezza media (in colonne), errore medio, throughput medio e sovrastima media.

Osservando la Tabella 7 si osserva che mediamente, le configurazioni della *Count-Min Sketch* doppia, a parità di spazio occupato in memoria, hanno un errore minore rispetto a quelle della *Count-Min Sketch* semplice. Ad esempio, la configurazione che occupa meno di $1.25Kb$ ha un errore medio di 0.18, inferiore a quello della *Count-Min Sketch* semplice che è pari a 0.37. L'unica eccezione si osserva per la configurazione che occupa tra 2.26 e $3.25Kb$, dove l'errore medio della *Count-Min Sketch* doppia è pari a 0.0040, leggermente superiore a quello della *Count-Min Sketch* semplice che è pari a 0.0034, anche se la sovrastima media è inferiore di un circa 100 elementi. In generale, la *Count-Min Sketch* risulta essere più leggera della *Count-Min Sketch* doppia, soprattutto per le configurazioni più grandi. Il throughput medio della *Count-Min Sketch* doppia risulta essere inferiore a quello della *Count-Min Sketch* semplice, con un valore medio di circa 1.3×10^6 elementi al secondo, contro i 2.2×10^6 elementi al secondo della *Count-Min Sketch* semplice.

Analisi dei tempi di esecuzione

Una ultima metrica da considerare è il tempo di esecuzione delle due strutture.

CMS 1	Min.	1° qt.	Mediana(IQR)	Media(sd.)	3° qt.	Max.
< 1.25	0.419	0.448	0.486(0.128)	0.507(0.066)	0.576	0.642
1.25-2.25	0.417	0.433	0.439(0.029)	0.447(0.020)	0.462	0.500
2.26-3.25	0.413	0.434	0.439(0.021)	0.445(0.017)	0.455	0.496
3.26-4.25	0.416	0.432	0.440(0.028)	0.446(0.019)	0.460	0.494
> 4.25	0.418	0.438	0.454(0.038)	0.459(0.026)	0.476	0.539
CMS 2	**Min.**	**1° qt.**	**Mediana(IQR)**	**Media(sd.)**	**3° qt.**	**Max.**
< 1.25	0.597	0.720	0.848(0.18)	0.816(0.10)	0.895	1.154
1.25-2.25	0.593	0.662	0.824(0.18)	0.769(0.10)	0.846	0.957
2.26-3.25	0.602	0.662	0.818(0.18)	0.761(0.09)	0.838	0.889
3.26-4.25	0.604	0.647	0.817(0.19)	0.755(0.10)	0.838	0.972
> 4.25	0.605	0.657	0.809(0.18)	0.753(0.09)	0.838	0.893

Tabella 8: Tabella di sintesi del tempo di esecuzione in secondi di una *Count-Min Sketch* semplice (`CMS 1`) e di una *Count-Min Sketch* doppia (`CMS 2`) di profondità 3 al variare della larghezza.

Come si può osservare dalla Tabella 8 le due *Count-Min Sketch* hanno tempi di esecuzione molto simili, con la *Count-Min Sketch* doppia che risulta essere leggermente più lenta della semplice. Il tempo medio per la *Count-Min Sketch* semplice è di circa 0.45 secondi per tutte le configurazioni, mentre per la *Count-Min Sketch* doppia è di circa 0.75 secondi. Tempo medio e tempo mediano nella *Count-Min Sketch* semplice sono molto vicini, al contrario, nella *Count-Min Sketch* doppia si osserva una maggiore differenza tra i due valori. Questo potrebbe suggerire una distribuzione asimmetrica del tempo per la *Count-Min Sketch* doppia.

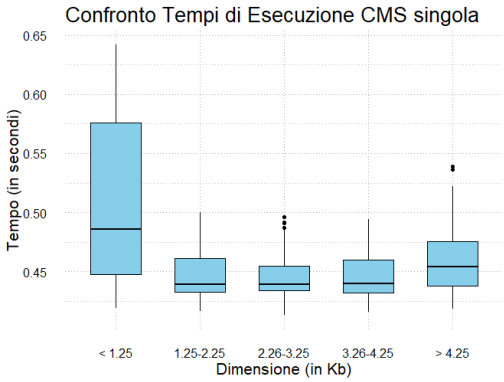

Figura 14: Boxplot dei tempi di esecuzione delle *Count-Min Sketch* singola.

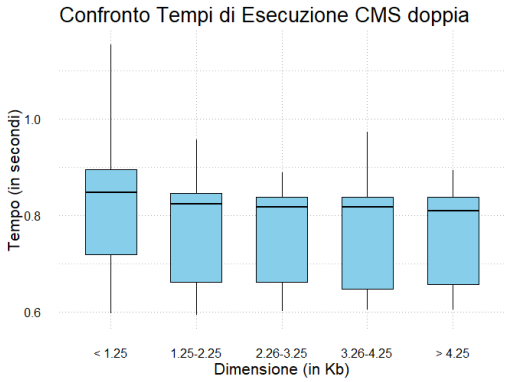

Figura 15: Boxplot dei tempi di esecuzione delle *Count-Min Sketch* doppia.

Dai boxplot nelle Figure 14 e 15, si può osservare che, a differenza dei boxplot della distribuzione uniforme, in Figura 7 e 8, le distribuzioni dei tempi di esecuzione presentano meno valori anomali. Il boxplot della *Count-Min Sketch* semplice per configurazioni con memoria inferiore a $1.25Kb$ si presenta più allungato rispetto agli altri, suggerendo una maggiore variabilità nei tempi di esecuzione. Le altre 4 configurazioni presentano boxplot molto simili tra loro. Appaiono schiacciati, suggerendo una distribuzione più uniforme dei tempi di esecuzione. Invece, i boxplot della *Count-Min Sketch* appaiono più allungati, con una mediana molto vicina al terzo quartile, suggerendo una distribuzione asimmetrica dei tempi di esecuzione, come già osservato in Tabella **??**.

4.3 Dataset `CIC-DDoS2019`

L'ultimo dataset in analisi, il `CIC-DDoS2019`, è un dataset di valutazione per gli attacchi *Distributed Denial of Service (DDoS)* sviluppato dal Canadian Institute for Cybersecurity dell'Università del New Brunswick. Il dataset è stato creato al fine di fornire un benchmark per la valutazione delle tecniche di rilevamento degli attacchi *DDoS* e include una varietà di attacchi *DDoS*, concentrandosi sugli attacchi che possono venire eseguiti usando i protocolli *TCP* e *UDP*. Questi attacchi possono essere di due tipi, sia *reflection based* che *exploration based*. Il dataset include anche traffico di rete benigno, che

può essere utilizzato per valutare la capacità delle tecniche di rilevamento di distinguere tra traffico benigno e attacchi *DDoS*. Per testare la struttura dati in analisi, si è deciso di utilizzare un attacco del tipo *UDP flooding*, in quanto rappresenta il tipo di attacco più comune.

4.3.1 Scelta della proporzione di divisione delle colonne della configurazione

Si procede ora a verificare, per varie configurazioni di colonne, quale proporzione di divisione meglio si adatta a questo dataset. Per la verifica è stato usato un dataset di training, sempre fornito dal *CIC*, contenente circa 1 milione di indirizzi *IP* e informazioni sul traffico di rete. In questa applicazione, ci concentreremo sugli indirizzi *IP* di origine, in quanto principale vettore di attacco per l'*UDP flooding*, che mira a sovraccaricare la rete della vittima con pacchetti *UDP* provenienti da molteplici fonti, ma la struttura dati può essere adattata per gestire anche altre tipologie di attacchi e di fonti di attacco. Sono state usate le stesse configurazioni di dimensione della sezioni precedenti: una configurazione da $1Kb$, una da $2Kb$ e una da $5Kb$.

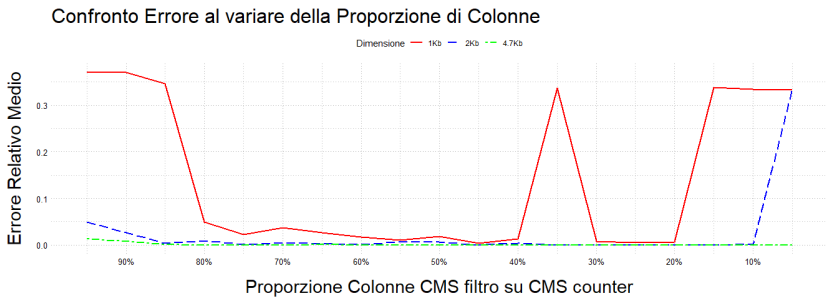

Figura 16: Evoluzione dell'errore in funzione della proporzione di divisione delle colonne su dataset `CIC-DDoS2019`.

Dalla Figura 16 si osserva come nella prima configurazione da $1Kb$, siano presenti valori molto alti di errore alle estremità, mentre al centro si osserva un errore medio contenuto, tranne che per un picco in corrispondenza del 35%. Il valore più basso si osserva nella configurazione in cui la *Count-Min Sketch* filtro ha un proporzione del 40% rispetto alla *Count-Min Sketch* conteggio. Per la configurazione da $2Kb$, l'errore medio minore si osserva un errore

medio molto basso per quasi tutte le proporzioni, con un picco di errore in corrispondenza della proporzione del 5%. La configurazione da $5Kb$ mantiene un errore prossimo allo zero per tutte le proporzioni. Anche in questo caso, si osserva come il valore di errore medio più basso per entrambe le configurazioni si osserva in corrispondenza dei valori centrali.

4.3.2 Analisi dei risultati dataset `CIC-DDoS2019`

Si passa ora a verificare le prestazioni della *Count-Min Sketch* doppia su questo dataset. Dai risultati ottenuti nella sezione precedente, e della Sezione *4.2.1*, si è deciso di utilizzare una divisione del 30% per le colonne addette filtraggio e al 70% per quelle dedite il conteggio. Sono state create diverse configurazioni della *Count-Min Sketch* doppia, con un numero di colonne che varia da $1Kb$ a $5Kb$, e sono state eseguite 100 simulazioni per ogni configurazione. La stessa configurazione, come fatto già in precedenza, è stata provata anche su una *Count-Min Sketch* classica, per poter confrontare i risultati. L'obbiettivo è identificare i due elementi più frequenti. Il primo elemento più frequente rappresenta la fonte dell'attacco, il secondo rappresenta il secondo indirizzo IP più frequente, che non è coinvolto nell'attacco, ma è comunque monitorato.

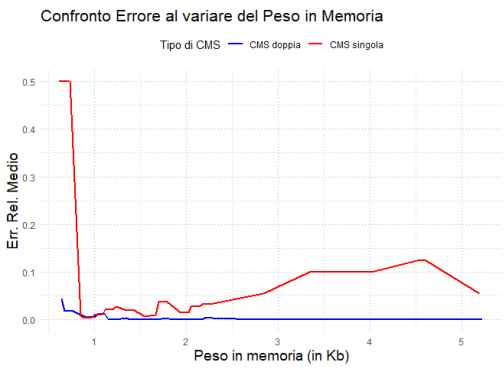

Figura 17: Evoluzione dell'errore in funzione della memoria in *KiloByte* su dataset `CIC-DDoS2019`.

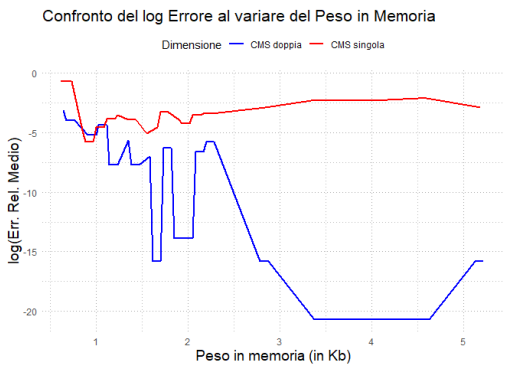

Figura 18: Evoluzione dell'errore su scala logaritmica in funzione della memoria in *KiloByte*.

Dalla Figura 17 si può osservare come l'errore medio della *Count-Min Sketch* doppia sia sempre molto vicino allo zero per tutte le simulazioni. La

Count-Min Sketch doppia risulta essere più precisa della *Count-Min Sketch* semplice per tutte le configurazioni, con errore quasi nullo. La controparte semplice, invece, presenta un errore medio molto maggiore rispetto alla *Count-Min Sketch* doppia, in particolare superato la soglia dei $2Kb$, l'errore della *Count-Min Sketch* semplice, fino a quel momento in diminuzione, torna ad aumentare.

Accuratezza nelle stime

Una prima analisi della precisione della struttura dati in analisi può essere effettuata confontando il gruppo degli *Heavy Hitters* individuati dalla *Count-Min Sketch* con il gruppo di *Heavy Hitters* reali.

Memoria (KB)	< 1.25	1.25-1.50	1.51-1.75	1.76-2.00	> 2.0
N. di HH	1.74	1.92	1.96	2.0	2.0
Precisione	0.87	0.96	0.98	1.00	1.00

Tabella 9: Numero di *Heavy Hitters* correttamente identificati e precisione (in percentuale) della *Count-Min Sketch* doppia.

Dalla Tabella 9 si può osservare come la *Count-Min Sketch* doppia riesca ad identificare correttamente tutti e 2 gli *Heavy Hitters* in tutte le sue configurazioni con memoria superiore ai $1.75Kb$. Le altre configurazioni comunque presentato una ottima accuratezza. Il valore più basso lo si osserva nelle strutture con memoria inferiore a $1.25Kb$ hanno una precisione dell'87%. Questo risultato è evidenziato anche dal basso errore in Figura 17.

Metriche di Errore

Dopo aver analizzato l'accuratezza delle stime della *Count-Min Sketch* doppia, si passa ad analizzare alcune metriche di errore.

CMS semplice	Dimensione	N. Col	Errore	Throughput	Sovrastima
< 1.25	0.94	32	0.28	8.79 $\times 10^4$	27
1.25-2.25	1.36	67	0.17	8.81 $\times 10^4$	43
2.26-3.25	1.59	87	0.28	8.79 $\times 10^4$	36
3.26-4.25	1.88	111	0.21	8.82 $\times 10^4$	54
> 4.25	3.39	240	0.13	8.74 $\times 10^4$	130

Tabella 10: Tabella di sintesi della *CMS* semplice di profondità 3 al variare della larghezza. Sono riportati i valori medi di dimensione (in Kb), larghezza media (in colonne), errore medio, throughput medio e sovrastima media.

Osservando la Tabella 10 si possono notare alcune statistiche delle varie configurazioni della *Count-Min Sketch* semplice. Le configurazioni che occupano meno di $1.25Kb$ hanno una dimensione media di 32 colonne, un errore medio di 0.28 e una sovrastima media di 27 elementi. Le configurazioni che occupano tra 1.25 e $1.50Kb$ hanno una dimensione media di 67 colonne, un errore medio di 0.17 e una sovrastima media di 43 elementi. Le configurazioni che occupano tra 1.51 e $1.75Kb$ hanno un errore medio di 0.28, pari alle strutture con memoria minore di $< 1.25Kb$, e una sovrastima di 36 elementi. Le configurazioni da 1.76 a $2Kb$ hanno una sovrastima leggermente maggiore, pari a 54 elementi, con un errore medio leggermente minore del precedente. Infine, per le strutture che occupano più di $2Kb$ si osserva un aumento della sovrastima media a 130 elementi e un errore medio pari a 0.13. In generale, si può osservare il trend già visto in Figura 17, ovvero che l'errore medio aumenta nuovamente per le configurazioni più grandi.

CMS doppia	Dimensione	N. Col	Errore	Throughput	Sovrastima
< 1.25	0.97	31	0.15	4.07 $\times 10^4$	28.46
1.25-2.25	1.74	97	0.042	4.07 $\times 10^4$	5.70
2.26-3.25	2.55	165	0.0011	4.07 $\times 10^4$	2.30
3.26-4.25	3.82	275	0.00049	4.06 $\times 10^4$	1.22
> 4.25	5.00	375	$\sim$0.0	4.05 $\times 10^4$	0.09

Tabella 11: Tabella di sintesi della *CMS* doppia di profondità 3 al variare della larghezza. Sono riportati i valori medi di dimensione (in *Kb*), larghezza media (in colonne), errore medio, throughput medio e sovrastima media.

Come riportato in Tabella 11, la *Count-Max Sketch* doppia risulta essere più pesante, in particolare le strutture più grandi, rispetto alla *Count-Min Sketch* semplice. In termini di errore medio, come già evidenziato in Figura 17, la *Count-Min Sketch* doppia risulta essere più precisa della controparte semplice per tutte le configurazioni. Le configurazioni più leggere hanno un errore medio che è quasi la metà di quello delle configurazioni della *Count-Min Sketch* semplice a parità di dimensione, nonostante una sovrastima simile. Anche i valori di sovrastima sono molto più contenuti. Ad esempio le strutture più pesanti hanno una sovrastima media di circa una unità, che è nettamente inferiore rispetto alle sovrastime della *Count-Min Sketch* semplice. In termini di throughput medio, la *Count-Min Sketch* doppia risulta essere peggiore rispetto alla semplice, che mostra un throughput superiore di quasi il doppio rispetto alla controparte doppia.

Analisi dei tempi di esecuzione

Una ultima metrica da considerare è il tempo di esecuzione delle due strutture.

CMS 1	Min.	1° qt.	Mediana(IQR)	Media(sd.)	3° qt.	Max.
< 1.25	1.090	1.115	1.131(0.032)	1.137(0.037)	1.147	1.375
1.25-2.25	1.099	1.116	1.131(0.030)	1.135(0.027)	1.146	1.250
2.26-3.25	1.095	1.115	1.130(0.031)	1.133(0.024)	1.146	1.219
3.26-4.25	1.085	1.115	1.131(0.033)	1.134(0.023)	1.148	1.185
> 4.25	1.096	1.122	1.143(0.039)	1.144(0.029)	1.161	1.264
CMS 2	**Min.**	**1° qt.**	**Mediana(IQR)**	**Media(sd.)**	**3° qt.**	**Max.**
< 1.25	2.387	2.421	2.448(0.057)	2.460(0.059)	2.479	2.824
1.25-2.25	2.386	2.431	2.448(0.048)	2.457(0.040)	2.479	2.625
2.26-3.25	2.402	2.431	2.447(0.045)	2.456(0.040)	2.476	2.637
3.26-4.25	2.403	2.434	2.463(0.049)	2.465(0.041)	2.483	2.611
> 4.25	2.399	2.434	2.457(0.058)	2.469(0.051)	2.492	2.648

Tabella 12: Tabella di sintesi del tempo di esecuzione in secondi di una *Count-Min Sketch* semplice (`CMS 1`) e di una *Count-Min Sketch* doppia (`CMS 2`) di profondità 3 al variare della larghezza.

Come si può osservare dalla Tabella 12 la *Count-Min Sketch* doppia risulta essere più lenta della *Count-Min Sketch* semplice in tutte le configurazioni di più di 1 secondo. I tempi minori per la *Count-Min Sketch* semplice sono in media sempre inferiori ai 1.2 secondi e non sembrano essere presenti dei trend legati alla dimensione. Per la *Count-Min Sketch* doppia, i tempi di esecuzione sono mediamente sono circa 2.4 secondi, la variabilità è maggiore rispetto alla versione semplice. Nonostante questo, la mediana e il tempo medio sono molto vicini, suggerendo una distribuzione simmetrica del tempo di esecuzione.

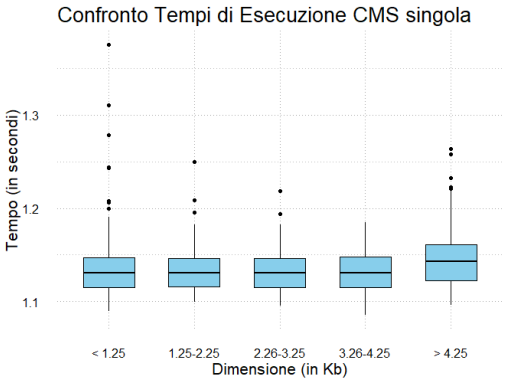

Figura 19: Boxplot dei tempi di esecuzione delle *Count-Min Sketch* singola.

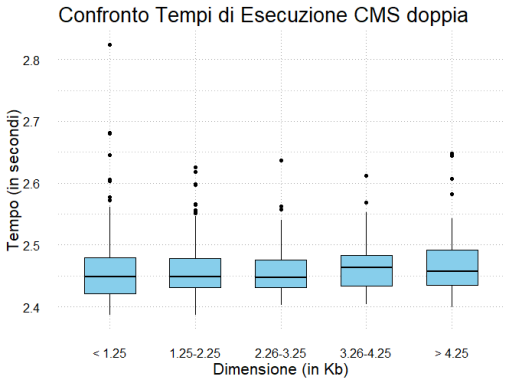

Figura 20: Boxplot dei tempi di esecuzione delle *Count-Min Sketch* doppia.

Le distribuzioni dei tempi di esecuzione, come mostrato nelle Figure 19 e 20, risultano avere valori anomali in quasi tutte le configurazioni, in particolare quelle con utilizzo della memoria $< 1.25Kb$ e per quelle con utilizzo della memoria $> 4.25Kb$ in entrambi i tipi di *Count-Min Sketch*. Anche se media e mediana sono molto vicine, la presenza di valori anomali indica che in alcune simulazioni il tempo di esecuzione è risultato essere significativamente più alto rispetto alla maggior parte delle altre simulazioni. Questo sta ad indicare come alcune configurazioni siano più sensibili a variazioni nei dati in ingresso, che porta il loro tempo di esecuzione a risultare rallentato. La configurazione più stabile in termini di tempo di esecuzione sembra essere quella con utilizzo della memoria compreso tra 3.26 e $4.25Kb$, che non presentano un elevato numero di valori anomali, in particolare nella *Count-Min Sketch* semplice.

5 Conclusioni

In questo lavoro, è stato proposto un algoritmo per il conteggio di eventi frequenti da uno stram di dati. L'algoritmo, basato su una *Count-Min Sketch* doppia ha portato a risultati promettenti, anche se altalenanti. L'obbiettivo principale era quello di identificare gli elementi frequenti e, quando applicato ad un dataset contenente dati da un attacco *DDoS*, ha mostrato la capacità di rilevare i picchi di traffico in modo efficace. Anche quando applicato ad un dataset uniforme, l'algoritmo ha mostrato buone prestazioni, suggerendo

anche la sua applicazione in scenari diversi e non per forza legati allo streaming. Ma, quando è stato testato su un dataset contenente dati da una distribuzione Zipf, ha mostrato difficoltà non nel distinguere tra gli elementi frequenti e quelli rari, ma nello stimarne la frequenza. Questi problemi sono comunque apparsi in configurazioni con un peso sulla memoria compreso tra $1Kb$ e $3Kb$, mentre configurazioni con un peso maggiore, non hanno mostrato questi cali di prestazioni, rimandendo in grado di stimare gli elementi frequenti. Possibili miglioramenti della struttura dati potrebbero includere l'uso di funzioni di hash più sofisticate, l'uso di un filtro migliore, la scelta del numero di *Heavy Hitters* senza bisogno di un input esterno, o ancora l'ottimizzazione della gestione della memoria, in particolare nel bilanciare le dimensioni delle due strutture *Count-Min Sketch* che compongono la struttura.

Riferimenti bibliografici

Lada Adamic and Bernardo Huberman. Zipf's law and the internet. *Glottometrics*, 3, 11 2001.

Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 20–29. Association for Computing Machinery, 1996. ISBN 0897917855. doi: 10.1145/237814.237823. URL https://doi.org/10.1145/237814.237823.

David A. Bader. *Massive Graph Analytics*. Chapman and Hall/CRC, May 2022. ISBN 9781003033707. doi: 10.1201/9781003033707. URL http://dx.doi.org/10.1201/9781003033707.

L. Breslau, Pei Cao, Li Fan, Graham Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings of the Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 126 – 134, 04 1999. ISBN 0-7803-5417-6. doi: 10.1109/INFCOM.1999.749260.

Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, April 2005. ISSN 0196-6774. doi: 10.1016/j.jalgor.2003.12.001. URL https://doi.org/10.1016/j.jalgor.2003.12.001.

Piotr Indyk, Frederik Mallmann-Trenn, Slobodan Mitrović, and Ronitt Rubinfeld. Online page migration with ml advice, 2020. URL https://arxiv.org/abs/2006.05028.

Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures, 2018. URL https://arxiv.org/abs/1712.01208.

Silvio Lattanzi, Thomas Lavastida, Benjamin Moseley, and Sergei Vassilvitskii. Online scheduling via learned weights, 12 2020.

Thodoris Lykouris and Sergei Vassilvitskii. Competitive caching with machine learned advice, 2020. URL https://arxiv.org/abs/1802.05399.

Michael Mitzenmacher and Sergei Vassilvitskii. Algorithms with predictions, 2020. URL https://arxiv.org/abs/2006.09123.

S. W. Roberts. Control chart tests based on geometric moving averages. *Technometrics*, 42(1):97–101, February 2000. ISSN 0040-1706. doi: 10.2307/1271439. URL https://doi.org/10.2307/1271439.

Gollapudi Sreenivas and Panigrahi Debmalya. Online algorithms for rent-or-buy with expert advice, 2019. URL https://proceedings.mlr.press/v97/gollapudi19a.html.

Kapil Vaidya, Eric Knorr, Tim Kraska, and Michael Mitzenmacher. Partitioned learned bloom filter, 2020. URL https://arxiv.org/abs/2006.03176.

Mingyuan Zang, Federico De Iaco, Jie Wu, and Marco Savi. In-kernel traffic sketching for volumetric ddos detection. *Proceedings of the IEEE International Conference on Communications (ICC)*, 2025.

Petar Čisar and Sanja Čisar. Ewma statistic in adaptive threshold algorithm. pages 51 – 54, 06 2007. ISBN 1-4244-1147-5. doi: 10.1109/INES.2007.4283671.