Università del Salento

Dipartimento di Ingegneria dell'Innovazione

Corso di Laurea Magistrale in Ingegneria Informatica

DOCUMENTAZIONE DI PROGETTO

Corso di Data Mining e Machine Learning

Implementazione dell'algoritmo di approssimazione dei quantili "EazyQuantile" in linguaggio C

Docenti

Prof. Massimo Cafaro

Autore

Culcea Cezar Narcis

Matricola n° 20086225

ANNO ACCADEMICO 2023/2024

Indice

Capitolo 1

Abstract

L'analisi dei **quantili** è uno strumento cruciale nello studio dei flussi di dati, poiché permette di esaminare la distribuzione effettiva dei dati. A differenza della media, che può mascherare picchi e variazioni significative, i quantili forniscono una visione più dettagliata e accurata delle caratteristiche dei dati.

Tra le varie applicazioni, una delle più critiche è la **QoS** nell'ambito delle Reti. Il monitoraggio delle metriche relative alle latenze o al throughput può fornire indicazioni sullo stato della congestione. In particolare molti **SLA Service Level Agreement** prevedono specifiche sul percentile 95° o 99°.

Per esempio se avessimo un una WebApplication con mediamente 100.000 richieste al giorno, una latenza media di 100 ms potrebbe sembrare adeguata, ma un 99° percentile di 500 ms indica che 1.000 richieste subiscono ritardi significativi, compromettendo l'esperienza utente e aumentando il rischio di abbandono.

La misurazione di queste metriche è da attribuire ai device di rete, che hanno un hardware limitato e non possono impiegare eccessive risorse.

Una possibile soluzione è l'algoritmo **EazyQuantile** [1], il quale approssima i quantili con una buona precisione, nonostante le poche risorse utilizzate.

Capitolo 2

Fondamenti

L'algoritmo soggetto della trattazione, esegue un calcolo per approssimare il quantile. Un quantile [2] di ordine $\alpha \in [0, 1]$ è un numero con valore q_α tale che divida la popolazione in due parti proporzionali a:

- α - popolazione minore di q_α
- $1 - \alpha$ - popolazione maggiore di q_α

I quantili il cui α è espresso tramite frazione hanno anche altri nomi, come il quantile 0.5 denominato mediana. Ovviamente per poter calcolare un quantile è necessario che il carattere della popolazione sia ordinato.

2.1 Naive Algorithm

Nel nostro caso d'uso, ci aspettiamo uno streaming di informazioni che rappresentano le metriche della QoS. Usando la definizione di quantile è possibile implementare il metodo naive di tracciamento dei quantili reali.

Esso consiste nell'accumulare i valori che arrivano man mano in una lista, ordinarli ed estrarre il valore corrispondente al quantile.

Questa soluzione è estremamente dispendiosa sia in fatto di tempo a causa del continuo ordinamento, sia in fatto di memoria (dato che lo stream è da considerarsi illimitato).

2.1.1 Pseudocodice

Algorithm 1 Calculate RealQuantile from stream

```
1:  procedure REALQUANTILE(nextItem, quantile)
2:      add nextItem to stream_data
3:      sort(stream_data)
4:      position ← quantile × (length(stream_data) −1)
5:      lowerIndex ← floor(position)
6:      upperIndex ← ceil(position)
7:      weight ← position − lowerIndex
8:      if weight == 0 then
9:          return stream_data[lowerIndex]
10:     else
11:         lowerValue ← stream_data[lowerIndex]
12:         upperValue ← stream_data[upperIndex]
13:         return lowerValue × (1 − weight) + upperValue × weight
```

2.1.2 Analisi Prestazioni

Notiamo che l'unica operazione con complessità diversa da $O(1)$ è l'operazione di sort che nel migliore dei casi non scende sotto $O(lg(n))$.

Considerando che inizialmente la lista è vuota e quindi ordinata, un problema alternativo potrebbe essere l'inserimento di un elemento in una lista ordinata, ma anche in questo caso non si scende sotto $O(lg(n))$ effettuando la ricerca della posizione tramite ricerca binaria.

Capitolo 3

EazyQuantile [3]

L'algoritmo si pone come obiettivo l'approssimazione del quantile scelto per uno stream di dati. Al fine di rendere più chiara la trattazione che seguirà nei capitoli successivi, elencherò i simboli della notazione che useremo:

- X - lo stream dei valori
- x_n - valore n-esimo
- p - quantile a cui siamo interessati
- q - valore del p-quantile di X
- q_n - valore stimato del p-quantile di X
- c_n^- - numero di valori dello stream minori di q_n
- c_n^+ - numero di valori dello stream maggiori di q_n
- sum_n - somma dei valori dello stream visti fino ad n
- avg_n - media dei valori dello stream visti fino ad n
- λ_n - passo di aggiornamento del quantile
- T - soglia della modalità di aggiornamento
- x_min - valore minore dello stream
- x_max - valore maggiore dello stream

L'idea base è sfruttare il conteggio dei valori minori e maggiori del quantile stimato per aggiornarlo in maniera incrementale attraverso due modalità:

- average - piccoli passi di aggiornamento (adatta ai quantili piccoli)
- min/max - grandi passi di aggiornamento (adatta ai quantili grandi)

3.1 Algoritmo

L'algoritmo si divide in 3 fasi principali:

1. selezione della modalità di aggiornamento
2. aggiornamento statistiche interne
3. aggiornamento del quantile stimato

Fase 1

Per ottenere il quantile consideriamo una funzione $g(x) : rank(x) \to x$ che assegna il rango di un valore al valore stesso. Sia il rango la posizione ordinata dei valori, abbiamo che su n valori, la posizione del quantile p è $n * p$. Di conseguenza $g(n * p)$ sarà il valore del quantile. All'arrivo del nuovo dato, la nuova posizione del quantile sarà $(n + 1) * p$ con valore $g((n + 1) * p)$. Di conseguenza il passo di aggiornamento tra i due quantili $\lambda_n = g((n+1)*p) - g(n*p)$. Il vero problema è approssimare la funzione $g(x)$ in maniera efficiente. L'algoritmo EazyQuantile risolve il problema introducendo due modalità di aggiornamento per approssimare λ_n.

La prima modalità è l'**average mode** ed approssima la funzione con curve lente. Basandoci sul principio delle sequenze aritmetiche si ha che

$$\lambda_n = 2 * (\sum_i x_i/n - x_{min})/(n-1) \approx 2 * avg_n/(n-1)$$

La seconda modalità di aggiornamento è **max-min**. Essa si aspetta grandi cambiamenti della pendenza di $g(x)$ per cui la sua pendenza viene stimata attraverso la differenza tra i valori minimi e massimi dello stream.

$$\lambda_n = (x_{max} - x_{min})/(n-1)$$

Dato che la stima della funzione $g(x)$ determina la precisione e la velocità di convergenza, una soglia T impostata a 0.7 assegna la modalità **average** ai quantili inferiori e la modalità **max-min** ai quantili più elevati.

Fase 2

Una volta selezionata la modalità di aggiornamento, è necessario aggiornare i contatori utilizzati per la stima dei quantili.

Al nuovo arrivo di un valore, esso viene confrontato con il quantile stimato q_n. Se il valore è minore, il contatore c_n^- viene incrementato di 1, altrimenti si incrementa il contatore c_n^+. E' importante notare che se i due contatori corrispondono esattamente ai valori $n*p$ e $n*(1-p)$, vuol dire che la posizione del quantile stimato corrisponde a quella del quantile reale e si ha $q_n = q$.

Fase 3

Durante l'aggiornamento dei contatori può succedere che non valga più la condizione $c_n^- + 1 < n*p$ o $c_n^+ + 1 < n*(1-p)$. In questo caso abbiamo la conferma che il quantile stimato non è il grandtruth e va aggiornato utilizzando il valore λ_n.

Se il contatore degli elementi minori supera la soglia, significa che il quantile stimato è maggiore del quantile reale. In questo caso, il quantile deve essere ridotto sottraendo λ, e contemporaneamente va incrementato il contatore degli elementi maggiori.

D'altra parte, se il contatore degli elementi maggiori supera la soglia, il quantile stimato è inferiore a quello reale. In questo caso, il quantile deve essere aumentato aggiungendo λ, e va incrementato il contatore degli elementi minori.

3.2 Pseudocodice

Algorithm 2 eazyQuantile

```
1:  procedure EAZYQUANTILE(x_n, quantile)
2:      if p < T then
3:          sum_n ← sum_{n−1} + x_n
4:          λ_n ← 2 × sum_n/n
5:      else
6:          λ_n ← x_max − x_min
7:      λ_n ← λ_n/(n − 1)
8:      if x_n < q_n then
9:          if c_n^− + 1 > n × p then
10:             q_{n+1} ← q_n − λ_n
11:             c_n^+ ← c_n^+ + 1
12:         else
13:             c_n^− ← c_n^− + 1
14:     else
15:         if c_n^+ + 1 > n × (1 − p) then
16:             q_{n+1} ← q_n + λ_n
17:             c_n^− ← c_n^− + 1
18:         else
19:             c_n^+ ← c_n^+ + 1
20:     return q_{n+1}
```

3.3 Analisi teorica delle prestazioni

Considerando la complessità, notiamo che ogni step di aggiornamento non contiene cicli dipendenti dalla dimensione dello stream, a differenza dell'algoritmo naive che richiedeva l'ordinamento. L'algoritmo EZQ permette di ottenere il nuovo quantile in tempo $O(1)$ una volta che gli viene fornito un nuovo valore x_n.

Per quanto riguarda la memoria, l'algoritmo non ha necessità di memorizzare lo stream, ma conserva una struttura dati minima formata da 4 valori (sum_n, avg_n, c_n^-, c_n^+), oltre ai valori relativi a T, p, q_n, n, x_{min} e x_{max}. Anche in questo caso il guadagno in termini di memoria risulta enorme.

3.4 Implementazione

Il codice dell'algoritmo è interamente contenuto nel file `EazyQuantile.c`

La struttura dati usata per la stima è mostrata nello snippet seguente. Si è preferito realizzare una struttura leggermente più corposa in favore della leggibilità. In realtà alcuni parametri sono eliminabili in quanto ricavabili in maniera indiretta dagli altri (es. observed_num = counter_low + counter_high)

```
typedef enum { MAX_MIN, AVERAGE } Mode;


typedef struct {
    double quantile;           /**< Quantile utilizzato per il calcolo */
    double estimate_value;     /**< Valore stimato corrente */
    double sum_value;          /**< Somma dei valori osservati */
    double max_value;          /**< Valore massimo osservato */
    double min_value;          /**< Valore minimo osservato */
    int observed_num;          /**< Numero totale di valori osservati */
    int counter_low;           /**< Contatore dei valori inferiori alla stima */
    int counter_high;          /**< Contatore dei valori superiori alla stima */
    double threshold;          /**< Soglia per il cambio di valore stimato */
    double lamb;               /**< Parametro di aggiornamento per la stima */
    double avg;                /**< Media dei valori osservati */
    double toggle_threshold;   /**< Soglia per il cambio della modalità di calcolo */
    Mode mode;                 /**< Modalità di calcolo (MAX_MIN o AVERAGE) */
} EZQ;
```

Una volta definita la struttura, sono state create due funzioni necessarie per allocarla, inizializzarla ed eliminarla. Notiamo che in fase di inizializzazione si sceglie modalità di aggiornamento del passo λ_n.

```
EZQ* create_EZQ(double quantile) {
    EZQ *ezq = (EZQ *)malloc(sizeof(EZQ));
    if (ezq == NULL) {
        fprintf(stderr, "Memory allocation failed!\n");
        exit(1);
```

```
    }
    ezq->quantile = quantile;
    ezq->estimate_value = 0;
    ezq->sum_value = 0;
    ezq->max_value = -1;
    ezq->min_value = DBL_MAX;
    ezq->observed_num = 0;
    ezq->counter_low = 0;
    ezq->counter_high = 0;
    ezq->threshold = 0;
    ezq->lamb = 0;
    ezq->avg = 0;
    ezq->toggle_threshold = 0.7;
    ezq->mode = (quantile > ezq->toggle_threshold) ? MAX_MIN : AVERAGE;
    return ezq;
}

void destroy_EZQ(EZQ *ezq) {
    free(ezq);
}
```

Infine è stata scritta la vera e propria logica di aggiornamento, seguendo lo pseudo-codice proposto nel capitolo precedente. La funzione `double update(EZQ *ezq, double stream_data)` si occupa di ricevere il nuovo valore dallo stream, aggiornare la struttura dati e calcolare il nuovo quantile stimato.

Inizialmente la funzione si occupa di aggiornare il contatore relativo al numero di osservazioni e calcola la soglia che servirà a far scattare l'aggiornamento in caso i contatori `low` e `high` diventino troppo grandi. La funzione inoltre imposta il quantile al valore ricevuto se questo è il primo dello stream. Successivamente aggiorna i campi relativi ai valori `min_value`, `max_value` e `sum_value`

```
    ezq->observed_num += 1;
    if (ezq->observed_num <= 1) {
        ezq->estimate_value = stream_data;
```

```
    return ezq->estimate_value;
}
ezq->threshold = ezq->observed_num * ezq->quantile;
if (stream_data < ezq->min_value) {
    ezq->min_value = stream_data;
} else if (stream_data > ezq->max_value) {
    ezq->max_value = stream_data;
}
ezq->sum_value += stream_data;
```

Il passo successivo è il calcolo del valore λ_n, effettuato con la seguente logica.

```
if (ezq->mode == AVERAGE) {
    ezq->lamb= 2*((ezq->sum_value/(ezq->observed_num*1.0))/(ezq->observed_num-1));
} else if (ezq->mode == MAX_MIN) {
    ezq->lamb = (ezq->max_value - ezq->min_value) / (ezq->observed_num - 1);
} else {
    fprintf(stderr, "No such mode!\n");
    exit(1);
}
```

Per ultimo avviene il calcolo vero e proprio del quantile e l'aggiornamento dei contatori, secondo la logica della terza fase spiegata nelle sezioni precedenti.

```
if (stream_data <= ezq->estimate_value) {
    if (ezq->counter_low + 1 > ezq->threshold) {
        ezq->estimate_value -= ezq->lamb;
        ezq->counter_high += 1;
    } else {
        ezq->counter_low += 1;
    }
} else if (stream_data > ezq->estimate_value) {
    if (ezq->counter_high + 1 > ezq->observed_num - ezq->threshold) {
        ezq->estimate_value += ezq->lamb;
        ezq->counter_low += 1;
```

```
    } else {
        ezq->counter_high += 1;
    }
}
```

Capitolo 4

Testing [4]

La simulazione è eseguita su un sistema con le seguenti specifiche:

- OS -> Ubuntu 22.04.04 LTS
- hardware -> AMD Ryzen 3 2200U

4.1 Esecuzione - script main.sh

Per facilitare il test è stato ideato uno script `main.sh` che permette di compilare il progetto ed eseguirlo con rapidità.

In particolare sono presenti le seguenti modalità di utilizzo:

- `./main.sh -s` -> Build del progetto (make all + pip install -r requirements.txt)
- `./main.sh -c` -> Rimuove i file compilati e le librerie installate dal requirements.txt
- `./main.sh -n` -> Testa con distribuzione normale
- `./main.sh -u` -> Testa con distribuzione uniforme
- `./main.sh -e` -> Testa con distribuzione esponenziale
- `./main.sh -z` -> Testa con distribuzione Zipfiana

I test con le varie distribuzioni seguono una logica comune:

1. Generare una sequenza secondo la distribuzione specificata della lunghezza `STREAM_LENGHT` specificata nello script (default 4096)

2. Fornire la sequenza in pipe al programma `./RealQuantile` per trovare il quantile `QUANTILE` specificato nello script (default 0.5)

3. Fornire la sequenza in pipe al programma `./EazyQuantile` per trovare il quantile `QUANTILE` specificato nello script (default 0.5)

4. Fornire gli array dei quantili reali e stimati al programma `plot.py` che si occuperà di mostrare la differenza tra la stima ed il groundtruth

Il programma `plot.py` è un piccolo script in Python utilizzato per eseguire il plot di due linee che mostrano l'andamento del valore del quantile man mano che arrivano informazioni dallo stream.

4.2 Esecuzione diretta

4.2.1 Generatori di distribuzioni

Al fine di testare il funzionamento sono stati implementati 4 generatori di stream:

- `UniformDistribution`: genera numeri appartenenti ad una distribuzione uniforme tra 0 e 1
- `NormalDistribution`: genera numeri appartenenti ad una distribuzione normale con media 0.5 e varianza 0.125
- `ExponentialDistribution`: genera numeri appartenenti ad una distribuzione esponenziale con $\lambda = 1.0$
- `ZipfianDistribution`: genera numeri appartenenti ad una distribuzione zipfiana con $s = 1.1$ su una popolazione di 100 elementi

Tutti i generatori seguono la seguente sinossi: `./[NomeDistribuzione] <STREAM_LENGHT>`

4.2.2 Calcolatori dei quantili

Sono stati implementati due modi di calcolo del quantile:

- `RealQuantile`: implementazione naive tramite sort del tracciamento del quantile
- `EazyQuantile`: implementazione dell'algoritmo EazyQuantile di tracciamento del quantile

Entrambi i programmi seguono la seguente sinossi ed aspettano dallo standard input gli elementi dello stream: `./[CalcolatoreQuantile] <QUANTILE>`

4.2.3 Esempio

E' possibile lanciare da solo il programma seguendo l'esempio mostrato nella seguente immagine.

```
kynesys@OldBoy:~/Scrivania/DataMining/Cezar Narcis Culcea$ ./NormalDistribution 10 | ./EazyQuantile 0.5
0.337763
0.337763
0.737590
0.737590
0.513132
0.513132
0.513132
0.513132
0.636760
0.636760
```

Figura 4.1: Esecuzione di EazyQuantile per cercare il quantile 0.5, dandogli in pipe uno stream che segue una distribuzione normale di 10 elementi

Il risultato viene stampato sul terminale dopo ogni aggiornamento.

4.3 Confronto quantili stimati e reali

E' possibile osservare la capacità di tracciamento dei quantili relative varie distribuzioni tramite il comando `./main.sh -u/-n/-e/-z`. Riporto di seguito l'output relativo alla ricerca del quantile 0.5 su uno stream di 4096 dati.

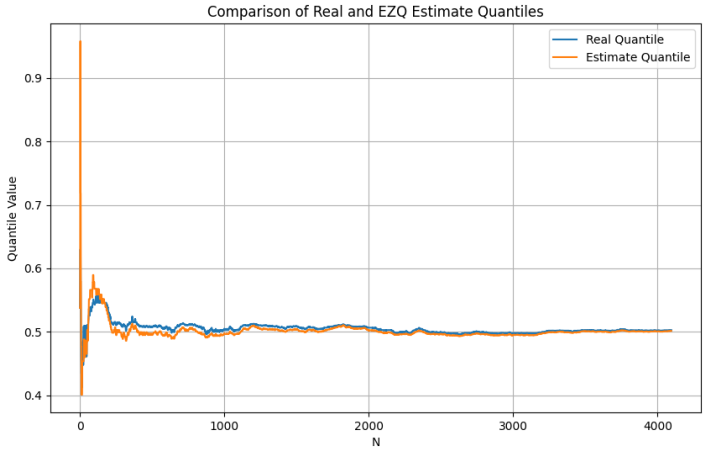

Figura 4.2: Tracciamento del quantile per la distribuzione Uniforme

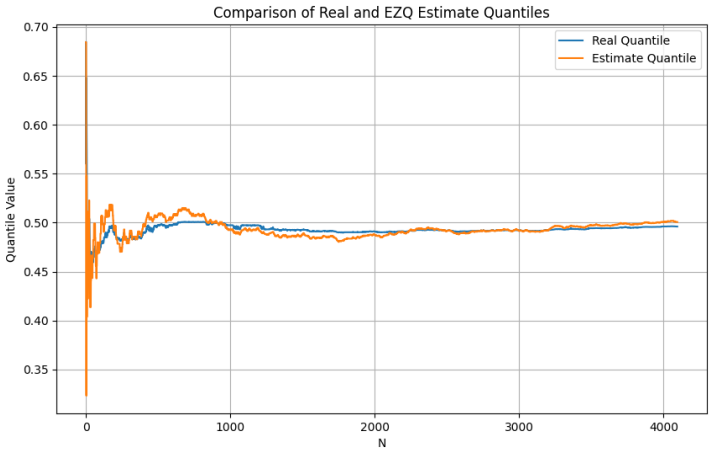

Figura 4.3: Tracciamento del quantile per la distribuzione Normale

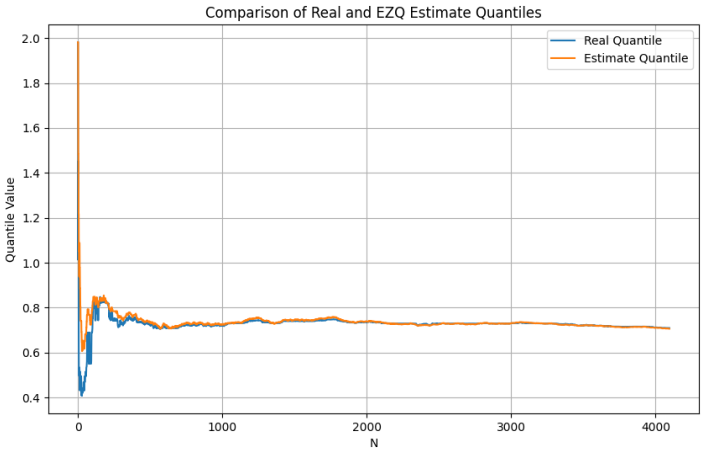

Figura 4.4: Tracciamento del quantile per la distribuzione Esponenziale

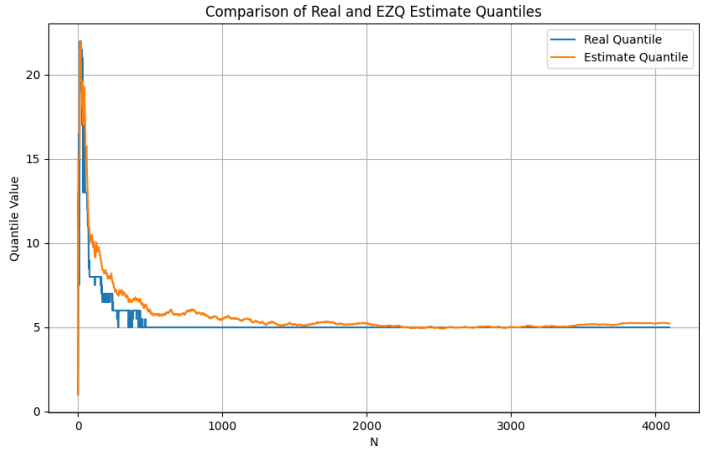

Figura 4.5: Tracciamento del quantile per la distribuzione Zipfiana

4.4 Benchmark - script measurements.sh

Al fine di misurare comodamente la precisione del tracciamento per tutte le distribuzioni è stato creato lo script `measurements.sh`. Esso viene lanciato senza opzioni ed utilizza i parametri definiti al suo interno:

- `STREAM_LENGHT`: lunghezza dello stream da generare
- `NUM_RUNS`: numero di run per ogni distribuzione, di cui fare la media
- `QUANTILES`: lista dei quantili che vogliamo analizzare
- `DISTRIBUTIONS`: elenco dei nomi dei generatori di distribuzioni da utilizzare

Il funzionamento è molto lineare. Per ogni distribuzione dati, esegue un numero di volte pari a `NUM_RUNS` la generazione dello stream. Per ogni stream traccia il quantile reale e stimato per ogni quantile. Prendendo il risultato finale stimato si confronta con il quantile finale reale e si calcola l'errore relativo con la seguente formula.

$$RelativeError = |q_n - q|/q$$

Infine per ogni quantile facciamo la somma di tutti gli errori relativi delle varie run e dividiamo per `MAX_RUNS` ottenendo *Average Relative Error*.

$$ARE = (\sum RelativeError)/MAX\_RUNS$$

Infine lo script sistema i dati per passarli al programma `measurements_plot.py` che si occupa di visualizzare su una scala logaritmica gli errori relativi medi che l'algoritmo EazyQuantile effettua in base al quantile cercato ed alla distribuzione in ingresso.

Lanciando il comando `./measurements.sh` otteniamo il seguente grafico.

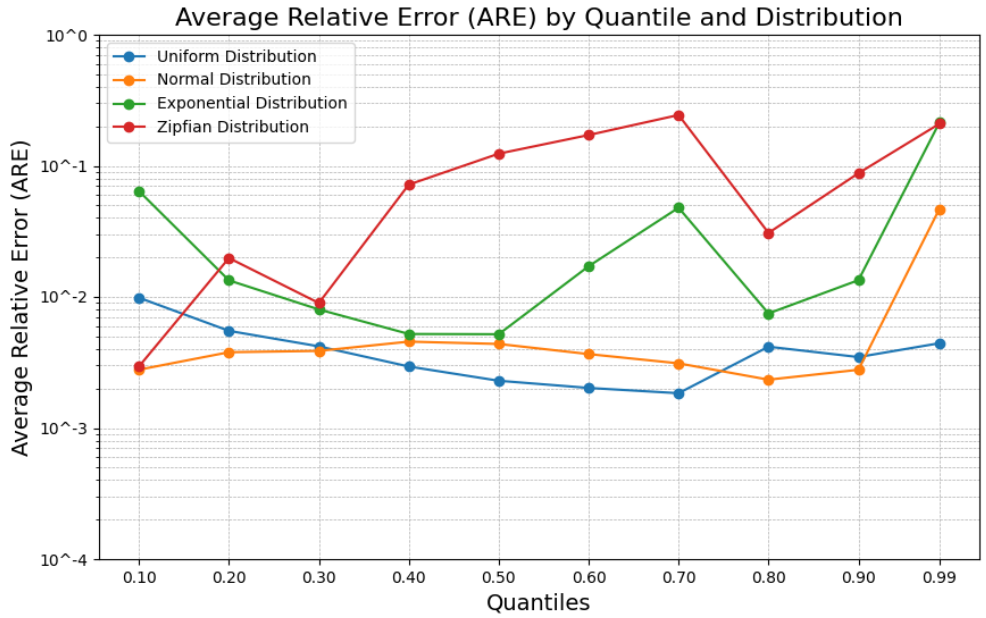

Figura 4.6: Errori relativi medi per quantili e distribuzioni

4.5 Esito degli esperimenti

Notiamo dalle singole run che l'algoritmo si comporta molto bene nel tracciamento del quantile, riuscendo quasi sempre ad essere molto vicino al quantile reale. Dai grafici si evince che al fine di ottenere dei valori precisi, l'algoritmo necessita di un valore compreso tra 1000 e 2000 dati al fine di riuscire a convergere verso il quantile reale.

Analizzando il grafico complessivo degli errori, notiamo come l'algoritmo si comporti molto bene per distribuzioni uniformi, normali ed esponenziali con errori dell'ordine di 10^{-2}. Esso ha però alcune difficoltà nei quantili più elevati (0.9, 0.99), in cui aumenta l'errore su tutte le distribuzioni. Per quanto riguarda la distribuzione zipfiana, l'algoritmo ha alcune difficoltà nei quantili centrali e finali, riportando un errore medio su tutti i quantili maggiore rispetto alle altre distribuzioni.

Capitolo 5

Conclusione

Gli errori più significativi emergono nella gestione di distribuzioni Zipfiane, che sono caratterizzate da una natura sbilanciata. Per esempio l'utilizzo delle parole di un vocabolario segue questa distribuzione (poche parole molto frequenti e molte parole poco frequenti). Questa lunga coda di eventi con bassa frequenza unita al picco di determinati elementi, rende difficile l'approssimazione dei quantili.

Nonostante ciò, l'algoritmo EazyQuantile dimostra complessivamente un'ottima performance. Esso riesce a stimare i quantili con una precisione notevole, considerando la velocità del singolo update O(1) e la memoria richiesta.

L'algoritmo è particolarmente efficace per le distribuzioni uniformi, normali ed esponenziali (tranne per i quantili più elevati). Per applicazioni che coinvolgono distribuzioni Zipfiane, sono consigliabili altri tipi di algoritmi, specialmente se vi sono stringenti requisiti sull'errore.

Bibliografia

[1] Lu Tang Bo Wang Rongqiang Chen. «EasyQuantile: Efficient Quantile Tracking in the Data Plane - Section 1». In: *School of Informatics, Xiamen University* (2023).

[2] Quantile - Wikipedia:'https://it.wikipedia.org/wiki/Quantile'.

[3] Lu Tang Bo Wang Rongqiang Chen. «EasyQuantile: Efficient Quantile Tracking in the Data Plane - Section 3». In: *School of Informatics, Xiamen University* (2023).

[4] Lu Tang Bo Wang Rongqiang Chen. «EasyQuantile: Efficient Quantile Tracking in the Data Plane - Section 4». In: *School of Informatics, Xiamen University* (2023).