UNIVERSITÀ DEL SALENTO

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

DATA MINING E MACHINE LEARNING

Docente: Massimo Cafaro

Big-Means: K-Means per il clustering dei Big Data

Autore: Emanuele Mele

Indice

Introduzione

Il problema del clustering è un problema NP-Hard che si occupa di comprendere la struttura intrinseca dei dati. A differenza dei problemi di classificazione o di regressione, dove, dato un set di punti di cui si conoscono gli attributi $X = (X_1, X_2, .., X_n)$ e il rispettivo valore di uscita Y, che risulta essere numerico nel caso di regressione e categorico nel caso della classificazione; nel clustering disponiamo unicamente di X. Il nostro scopo è quindi: dati dei punti di cui possiamo misurare una distanza, creare dei *cluster* che raggruppino i punti in modo tale che: i punti interni a un cluster siano molto simili tra loro e punti non appartenenti allo stesso cluster siano quanto più differenti possibile. Esistono due diverse categorie di algoritmi di clustering, gli algoritmi basati su appoccio *Point Assignment* oppure *Hierarchical*. Nel primo caso tendiamo a lavorare sul concetto di "vicinanza" dei punti a un determinato cluster, quindi facciamo un assegnazione basata sulla distanza tra punti. Il secondo caso si suddivide a sua volta in una metodologia chiamata *Agglomerative*, dove inizialmente ogni punto è un cluster e a ogni iterazione fondiamo i vari cluster attraverso delle metriche di coesione, oppure *Divisive* in cui facciamo il processo inverso, ovvero partiamo da un unico cluster che contiene tutti i punti dividiamo ricorsivamente seguendo un criterio di divisione. Questo scritto si concentra a raccontare e descrivere un algoritmo di tipo point assigment pensato per la manipolazione di grandi quantità di dati.

1 Clusterign di Big Data

Nei casi più semplici in cui il dataset è sufficientemente piccolo per essere elaborato in memoria esisto differenti algoritmi di clustering, di tipologia *Point Assignment* o *Hierarchical* che permettono di raggruppare i punti in modo omogeneo. Altri algoritmi nascono con l'obiettivo di trattare grandi quantità di dati, e quindi di effettuare iterativamente dei clustering su una porzione del dataset, un esempio è l'agoritmo BFR (Bradley Fayyad Reina) [1]. Nei casi più comuni gli algoritmi di clustering basati su un'approccio point assignment in cui devo minimizzare una funzione di loss che si riconduce spesso a una somma degli scarti quadratici, più precisamente una somma della norma euclidea al quadrato degli scarti. Quindi avendo $C = (c_1, c_2, .., c_k)$ centroidi (che rappresentano il cluster, più precisamente il suo centro) e $X = (x_1, x_2, .., x_m)$ punti io voglio:

$$\min_{\mathbf{c}} \sum_{i=1}^{m} \|\mathbf{x_i} - \mathbf{c_j}\|_2^2 \quad \forall j \in \{1, 2, \ldots, k\} \tag{1}$$

dove c rappresenta il centroide del j-esimo cluster. Per un generico k e m il problema è definito come NP-hard. Quando si va in contro a dati di dimensionalità elevata possiamo ricadere in quella che viene definita come *Curse of Dimensionality* in cui non siamo più in grado di discriminare il concetto di *neighborhood* e quindi approcci basati sulle distanze falliscono nel loro compito, per questo, quando parliamo di big data, faremo riferimento a dati che presentano un numero elevato di istanze ma con dimensionalità relativamente basse, quindi con un numero moderato di feature, in modo da non ricadere in situazioni sgradevoli. Lavorare su una grande quantità di dati implica necessariamente svolgere grandi quantità di operazioni e quindi aumentare i tempi di calcolo. In generale si cerca di mantenere un approccio euristico, semplice, che però spesso porta a soluzioni che rimangono valide solo in casi in cui il dataset è di dimensioni accettabili; uno degli algoritmi che ha riscontrato più successo a riguardo è l'algoritmo *K-Means* anche conosciuto come algoritmo di *Lloyd* [2], che utilizza una funzione di loss come quella citata in precedenza (1) per determinare dei cluster omogenei. Questo algoritmo necessita di tutti i punti del dataset per effettuare il clustering, quindi non è adatto a trattare una grande quantità di dati. Quello che faremo di seguito è vedere come utilizzare il K-Means per effettuare dei clustering parziali che ci permetteranno di determinare dei possibili centroidi accettabili per il dataset completo.

2 Premesse

Per comprende a pieno l'algoritmo di clustering Big-Means necessitiamo di svolgere alcune premesse illustrando gli algoritmi utilizzati da quest'ultimo. In particolare necessitiamo di spiegare il funzionamento di K-Means e K-Means++ per comprendere a pieno Big-Means

2.1 K-Means

Come anticipato nella sezione precedente l'algoritmo di clustering K-Means [2] si occupa di effettuare il clustering di k cluster utilizzando un approccio point assigment. Dato un dataset di punti $D = (x_1, .., x_n)$ vogliamo determinare $C = (C_1, .., C_k)$ cluster, rappresentati da k centroidi $c = (c_1, .., c_k)$ che sono dati dalla media dei punti appartenenti a un cluster:

$$c_j = \frac{1}{n_j} \sum_{x_i \in C_j} x_i \tag{2}$$

Quindi il centroide c_j è dato dalla somma dei punti x_i appartenenti al cluster C_j diviso il numero di punti n_j, che rappresentano il nuemro di punti del cluster j-esimo (2). Un punto x_i appartiene a un cluster C_j tale per cui la sua distanza dal centroide c_j è minima (4). Se utilizziamo come metrica di distanza la norma 2 al quadrato avrò che (3):

$$dist(\mathbf{x_i}, \mathbf{c_j}) = \|\mathbf{x_i} - \mathbf{c_j}\|_2^2 \tag{3}$$

Il centroide che sceglieremo c_s per il punto x_i sarà quello a distanza minima, quindi:

$$c_s = \arg\min_{\mathbf{j=1}}^{k} \{\|\mathbf{x_i} - \mathbf{c_j}\|_2^2\} \tag{4}$$

In generale noi vorremmo i cluster tali per cui minimizziamo la *sum of squared error*, ma come detto precedentemente il problema è NP-hard (5).

$$SSE(C) = \sum_{j=1}^{k} \sum_{x_i \in C_j} \|\mathbf{x_i} - \mathbf{c_j}\|_2^2 \tag{5}$$

Quello che possiamo fare è utilizzare un approccio greedy, dove scegliamo la soluzione ottima localmente a ogni iterazione, quindi, calcoleremo la distanza di un punto dai k centroidi e trovata la distanza minima rispetto all'i-esimo centroide assegneremo il punto al cluster i-esimo, questa operazione va effettuata per tutti i punti presenti nel dataset. Successivamente i cluster avranno un certo numero di punti, ciò implica che il centroide di ogni cluster va aggiornato ricalcolando la media dei punti dei cluster, tutto questo processo va iterato fino a convergenza. La convergenza è dettata da una soglia da noi scelta o dal numero di iterazioni, è ragionevole però seguire una logica di questo tipo:

$$\sum_{j=1}^{k} \|(c_j^t - c_j^{t-1})\|_2^2 \leq \epsilon \tag{6}$$

Dove c_j^t rappresenta il centroide j-esimo all'iterazione t. Quindi quello che facciamo è fermare l'algoritmo quando la somma della variazione della distanza tra i centroidi è inferiore o uguale a una soglia ϵ da noi selezionata (6). Riassumendo gli step da seguire saranno:

1. Inizializzare i primi k centroidi
2. Calcoliamo per ogni punto la distanza dai k centroidi
3. Assegniamo i punti ai cluster più vicini
4. Aggiorniamo i centroidi
5. Se non siamo arrivati a convergenza ripartiamo del punto 2

Ovviamente, come detto in precedenza, la scelta del numero di cluster k, l'inizializzazione dei centroidi e la scelta del parametro ϵ sono a nostra discrezione, quindi c'è alta variabilità dei risultati dettata dalle nostre scelte iniziali. L'inizializzazione dei centroidi può essere effettuata in modo pseudo-casuale, ovviamente il risultato finale è fortemente condizionato da questa scelta. Esisto metodi più performanti riguardanti l'inizializzazione dei primi k centroidi, uno di questi è l'algoritmo *K-Means++*, di cui ci occuperemo in seguito [3].

2.1.1 Pseudocodice K-Means

Si illustra di seguito lo pseudocodice dell'algoritmo K-Means basato su un inizializzazione dei centroidi random. In input indichiamo con D, k, ϵ rispettivamente il dataset di punti, il numero dei centroidi e la soglia di stop; in output abbiamo $C, SSE(C)$ che indicano i cluster finali e il valore della funzione di loss finale.

Algorithm 1: K-Means

Data: D,k,ϵ

Result: C,SSE(C)

Inizializzo casualmente i k centroidi $c_1, .., c_k$

$t = 0$

while $\sum_{j=1}^{k} \|(c_j^t - c_j^{t-1})\|_2^2 \leq \epsilon$ **do**

  $C_j = \emptyset \quad \forall j \in (1, 2, .., k)$

  for *each* $x_i \in D$ **do**

    $c_s = \arg\min_{\mathbf{j=1}}^{k} \{\|\mathbf{x_i} - \mathbf{c_j}\|_2^2\}$ //Mi determino il centroide più vicino

    $C_s = C_s \cup x_i$ //Assegno il punto al cluster più vicino

  for $j = 1$ *to* k **do**

    $c_j^t = \frac{1}{n_j} \sum_{x_i \in C_j} x_i$ //Aggiorno i centroidi

  $t = t + 1$

$SSE(C) = 0$

for $j = 1$ *to* k **do**

  $SSE(C) = \sum_{x_i \in C_j} \|\mathbf{x_i} - \mathbf{c_j}\|_2^2 + SSE(C)$

Di seguito vediamo come si comporta l'algoritmo nelle varie iterazioni impostando un valore di $k = 2$.

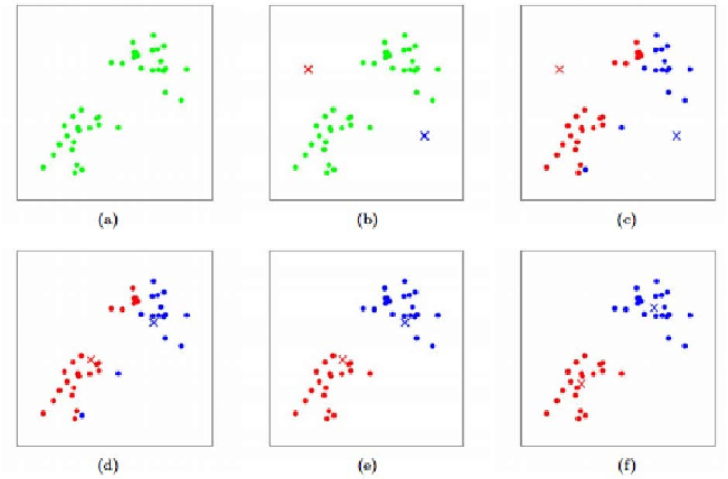

Figura 2.1: Esecuzione dell'algoritmo K-Means con $k = 2$, è possibile visualizzare le iterazioni fino a convergenza. [4]

Dalle immagini 2.1 è possibile ripercorrere ciò che abbiamo detto, partendo dalla scelta casuale dei due centroidi (b), l'assegnazione dei cluster ai punti (c), il ricalcolo dei centroidi (d), una fase ulteriore di assegnazione dei punti e ricalcolo (e) fino a convergenza (f).

2.2 K-Means++

Uno dei problemi durante l'utilizzo dell'algoritmo di clustering K-Means è l'inizializzazione dei centroidi. Nella versione base dell'algoritmo si effettua l'inizializzazione dei centroidi in modo pseudocasuale, questo può far variare notevolmente i cluster finali e inoltre potrebbero presentarsi dei problemi relativi alla convergenza dell'algoritmo. Per ovviare a questo problema quello che si fa è utilizzare *K-Means++* [3] che è un metodo di inizializzazione dove si sceglie solo il primo centroide in modo pseudocasuale, invece di sceglierne k casuali, successivamente alla scelta del primo si scelgono gli altri centroidi in modo che questi abbiano una probabilità di essere scelti proporzionale alla loro distanza rispetto ai centroidi precedenti. Più precisamente determinerò per ogni punto $x \in X$ con X dataset:

$$P(c_i = x) = \frac{\min_{j=1,..,i-1} \|\mathbf{x} - \mathbf{c_j}\|_2^2}{\sum_{x \in X} \min_{j=1,..,i-1} \|\mathbf{x} - \mathbf{c_j}\|_2^2} \tag{7}$$

Quello che vogliamo determinare in (7) è la probabilità che il nuovo centroide c_i sia il punto x, questa è data dalla distanza tra il punto x e il centroide più vicino c_j; tutto diviso la somma delle distanze di ogni punto rispetto al centroide più vicino. Una volta ottenute tutte le probabilità per ogni punto, andremo a selezionare un punto seguendo la PMF ricavata da queste valutazioni, quindi non prenderemo esattamente il punto più distante rispetto a tutti i centroidi precedenti, ma selezioneremo un punto in maniera casuale ma dando più probabilità di essere selezionati come nuovi centroidi ai punti più distanti.

2.2.1 Pseudocodice K-Means++

Andiamo ora a illustrare lo pseudocodice dell'algoritmo K-Means++ indicando con X il dataset e con k il numero di centroidi desiderati.

Algorithm 2: K-Means++

Data: X,k

Result: $C = \{c_1, .., c_k\}$

$C = \emptyset$

$C =$ (Inizializzo casualmente il primo centroide c_1) $\cup\, C$

for $i = 1$ *to* k **do**

  for $h = 1$ *to* $|X|$ **do**

    $P(c_i = x_h) = \frac{\min_{j=1,..,i-1} \|\mathbf{x_h} - \mathbf{c_j}\|_2^2}{\sum_{x \in X} \min_{j=1,..,i-1} \|\mathbf{x} - \mathbf{c_j}\|_2^2}$

  $C =$ (Seleziono c_i casualmente da X in accordo con la distribuzione di probabilità ricavata precedentemente) $\cup\, C$

Di seguito vediamo un esempio con $k = 4$ centroidi:

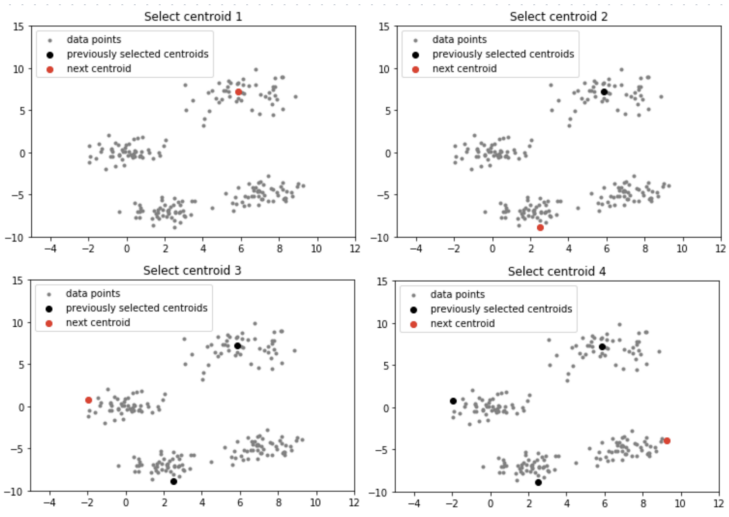

Figura 2.2: Esecuzione dell'algoritmo K-Means++ con $k = 2$ [5].

Nella prima iterazione 2.2 il centroide viene selezionato in maniera casuale, successivamente la scelta dei punti è fatta in modo tale che siano distanti dai precedenti centroidi, ovviamente bisogna tenere conto del fattore probabilistico, quindi non è certo che venga scelto il punto più distante in assoluto da tutti gli altri centroidi, ma è sicuramente più probabile che un punto lontano venga selezionato con più facilità.

3 Big-Means

L'algoritmo Big-Means ideato dai ricercatori Mussabayev, Mladenovic, Jarboui e Mussabayev [6] vuole risolvere il problema relativo al clustering dei big data utilizzando un tecnica di tipo point assigment. Anche in questo caso l'obiettivo ideale è quello di minimizzare la funzione di loss:

$$\min_{\mathbf{c}} \sum_{i=1}^{m} \|\mathbf{x_i} - \mathbf{c_j}\|_2^2 \quad \forall j \in \{1, 2, \ldots, k\} \tag{8}$$

Come sappiamo non è possibile trovare la soluzione esatta in tempi accettabili. Quello che faremo è servirci dell'algoritmo di clustering K-Means, utilizzando come metrica di distanza la distanza euclidea e come funzione di loss la SSE, illustrate anche precedentemente per il K-Means:

$$SSE(C) = \sum_{j=1}^{k} \sum_{x_i \in C_j} \|\mathbf{x_i} - \mathbf{c_j}\|_2^2 \tag{9}$$

L'algoritmo effettua il clustering su un campione di dati di dimensione $\{s \quad t.c \quad s << m\}$, con m dimensione del dataset, utilizzando K-Means, successivamente utilizza i centroidi ricavati dal clustering precedente per inizializzare i centroidi di K-Means su un nuovo campione di dati sempre di dimensione s, i centroidi restituiti da K-Means sono mantenuti solo se il clustering all'iterazione i-esima è migliorativo rispetto alle precedenti iterazioni. Tutta questa operazione viene ripetuta fino a quando non si raggiunge un criterio di stop da noi deciso. Quindi, dato un dataset D, a ogni iterazione i utilizzeremo un campione di dati $\{X_i \in D \quad t.c \quad |X_i| = s\}$, effettueremo il clustering del campione con K-Means, se i centroidi C trovati migliorano la funzione di loss f_{opt} delle iterazioni precedenti acquisisco i centroidi e itero nuovamente.

3.1 Pseudocodice Big-Means

Vediamo nel dettaglio come si comporta l'algoritmo. Indichiamo con X il dataset, con s la grandezza del campione e con k il numero di centroidi desiderati. In uscita vogliamo i centroidi $C = \{c_1, .., c_k\}$ che hanno ottenuto un punteggio migliore rispetto ai differenti cluster valutati.

Algorithm 3: Big-Means

Data: X,k,s

Result: $C = \{c_1, .., c_k\}$

$C = \emptyset$

$f_{opt} = \infty$

while *non si verifica la condizione di stop* **do**

- $P =$ campiono uniformemente s punti da X
- $C' = C$
- Reinizializzo i cluster degeneri (se presenti) utilizzando K-Means++
- $C'' = KMeans(P, C')$
- **if** $f_{opt} > f(C'', P)$ **then**
  - $C = C''$
  - $f_{opt} = f(C'', P)$

//OPZIONALE

$A = f(C, X)$ //Determino il punteggio della funzione obiettivo utilizzando i centroidi trovati e tutto il dataset

Necessitiamo di effettuare alcune precisazioni riguardanti le procedure che eseguite effettuate nello pseudocodice. Quando effettuiamo un clustering su un campione di dati possiamo trovarci in situazioni

in cui ricaviamo dei cluster degeneri, quindi dei cluster che contengono solo un punto, se questo dovesse succedere reinizializziamo il cluster utilizzando l'algoritmo K-Means++. È presente anche una parte opzionale, eseguibile unicamente su dei dataset di dimensione contenuta, dove andiamo a determinare il valore della funzione di loss rispetto ai centroidi trovati da Big-Means utilizzando tutti i punti del dataset, questa operazione risulta necessaria nel momento in cui vogliamo confrontare le prestazioni di questo algoritmo, che effettua un clustering basato sul campionamento, rispetto ad algoritmi come K-Means che effettuano il clustering su tutti i punti del dataset.

3.2 Analsi e valutazioni

Anche in questo algoritmo si pone il problema di inizializzare i centroidi alla prima iterazione, quello che si fa è utilizzare K-Means++ per l'inizializzazione e per l'eliminazione dei centroidi degeneri. Il campionamento degli s punti deve essere svolto nella maniera più veloce possibile, può essere eseguito in $O(1)$ se utilizziamo una costante s come dimensione del campione. La scelta delle dimensioni del campione è un fattore cruciale nell'algoritmo, all'aumentare di s la fluttuazione della soluzione dovrebbe ridursi poichè andremo a lavorare sempre con gli stessi dati, viceversa se s è piccola la soluzione tenderà a fluttuare maggiormente, questo poichè andremo a lavorare quasi sempre con valori differenti. Necessitiamo di cercare un compromesso sulla dimensione s in modo che la variabilità della soluzione non sia troppo elevata, ma allo stesso tempo non si perda quella che è la forma naturale dei dati nella loro interezza. L'algoritmo si presta ad essere parallelizzato, questo poichè possiamo e rendere indipendente l'esecuzione di K-Means su differenti campioni. Utilizzando l'algoritmo proposto con parametri appropriati, si possono ottenere risultati di clustering accettabili in tempi decisamente più rapidi. La complessità di Big-Means è $O(s \cdot n \cdot k)$ ad ogni iterazione e la complessità di K-Means++ è $O(m \cdot n \cdot k)$. La convergenza di Big-Means può essere raggiunta anche per una dimensione del campione relativamente piccola $s << m$, in questo modo il tempo di esecuzione risulta notevolmente minore rispetto a K-Means o K-Means++ sull'intero dataset.

3.3 Implementazione

Il linguaggio di programmazione scelto per l'implementazione di Big-Means è C++. Come detto precedentemente necessitiamo di implementare gli algoritmi K-Means e K-Means++, anche questi sono stati implementati utilizzando C++.
L'algoritmo K-Means è stato strutturando secondo lo pseudocodice indicato nel capito precedente 1. Le componenti principali, utilizzate anche dagli altri algoritmi, sono il calcolo della norma euclidea al quadrato tra due punti:

```
double SquaredOfDistances(vector<double> punto1, vector<double> punto2) {
// Assicurati che entrambi i vettori abbiano la stessa dimensione
if (punto1.size() != punto2.size()) {
    cout << "Errore: I vettori devono avere
    la stessa dimensione per poter calcolare la distanza uclidea.
    \nControlla di aver aperto correttamente il file!" << endl;
    exit(1);
    return -1.0;  // Valore di errore
}

double sum = 0;
for (int i = 0; i < punto1.size(); i++) {
    double diff = punto2[i] - punto1[i];
    sum += diff * diff;
}

return sum;
}
```

Come è possibile notare i punti sono stati considerati come dei vettori di double, quindi diamo per scontato che un punto contenga dei punti assimilabili solo a dei valori double; si è utilizzata la struttura *vector* così da lavorare con punti n dimensionali. Dopo una fase di assegnazione dei punti al cluster un'altra operazione principali svolta dal K-Means è l'aggiornamento dei centroidi:

```
vector<double> computeCentroid(vector<vector<double>> punti) {
// Verifica se il vettore di punti è vuoto
if (punti.empty()) {
    cerr << "Il vettore di punti è vuoto. Impossibile calcolare la media.\n";
    return {}; // Restituisci un vettore vuoto in caso di errore
}

// Inizializzazione del vettore risultante con la
stessa dimensione del primo punto
vector<double> media(punti[0].size(), 0.0);

// Somma di tutti i punti
for (const auto& punto : punti) {
    for (std::size_t i = 0; i < punto.size(); ++i) {
        media[i] += punto[i];
    }
}

// Calcolo della media dividendo per il numero di punti
for (double& valore : media) {
    valore /= punti.size();
}

return media;
}
```

Per il resto l'algoritmo itera questi processi fino a convergenza 6, per la nostra applicazione specifica abbiamo impostato il valore di $\epsilon = 0.0001$. L'inizializzazione dell'algoritmo Big-Means, e la gestione dei cluster degeneri, necessita dell'utilizzo di K-Means++. Come indicato precedentemente nello pseudocodice 2 una parte fondamentale dell'algoritmo è la scelta pseudocasuale del centroide basata sulla distribuzione di probabilità fornita dai dati in funzione delle distanze calcolate:

```
int estraiPunto(vector<double>& probabilita) {
// Verifica se il vettore delle probabilità è vuoto
if (probabilita.empty()) {
    cout << "Errore: Vettore delle probabilità vuoto." << endl;
    exit(1);
    return -1;  // Ritorna un valore speciale per indicare un errore
}

// Calcola la probabilità cumulativa
vector<double> prob_cumulativa = probabilita;
partial_sum(prob_cumulativa.begin(),
prob_cumulativa.end(), prob_cumulativa.begin());

// Genera un numero casuale tra 0 e la somma delle probabilità
random_device rd;
mt19937 gen(rd());
uniform_real_distribution<> dis(0, prob_cumulativa.back());
double random_value = dis(gen);

// Trova l'indice dell'elemento corrispondente
nella probabilità cumulativa
auto it = lower_bound(prob_cumulativa.begin(),
prob_cumulativa.end(), random_value);

// Restituisci l'indice estratto
return distance(prob_cumulativa.begin(), it);
}
```

Dopo aver implementato gli algoritmi K-Means e K-Means++ possiamo assemblarli per costruire Big-Means. Prima di procedere con la discussione dell'implementazione di Big-Means necessitiamo di trattare la tecnica utilizzata per leggere i nostri dataset. Sono state implementate due funzioni differenti per la lettura del dataset per dei motivi ben precisi. Una prima funzione più semplice e banale è quella che acquisisce tutto il dataset in memoria, questa funzione è utile nel momento in cui vogliamo effettuare dei benckmark, infatti, gli autori hanno seguito esattamente questa procedura: hanno determinato i centroidi con Big-Means utilizzando campioni del dataset, successivamente hanno determinato il valore finale della funzione di loss su tutti i punti del dataset in modo da confrontarlo con lo stato dell'arte. Nel caso reale l'algoritmo è pensato per lavorare con quantità di dati elevate, questo implica che non è possibile mantenere in memoria l'itero dataset, quindi si è optato per effettuare un campionamento direttamente dal file, quello che si fa è spostare in maniera casuale il file pointer in differenti righe in modo da acquisire esattamente gli s punti desiderati, questo metodo però pone l'attenzione a non acquisire più volte lo stesso punto. Alla fine del documento è illustrata la struttura che deve avere il dataset per essere letto correttamente dal nostro porgramma C++.

```
vector<vector<double>> readData(const string& fileName,
int headerRow,int labelColumn, char delimiter,
int* totalRow,int* columnNumber);

vector<vector<double>> openAndSampleData(const string& fileName,
int headerRow,int labelColumn, char delimiter,
int sample,int seed);
```

Analizziamo i diversi parametri che richiede la funzione:

- fileName: richiede una stringa con il nome del file relativo al dataset più la path, se necessaria
- headerRow: indica la riga da cui iniziare a leggere il file (nel caso in cui ci siano file con una o più righe di intestazione)
- labelColumn: indica il numero della colonna da eliminare se presente (nel caso in cui si vogliano mantenere tutte le colonne il valore viene impostato a -1)
- delimiter: indica il carattere separatore tra i valori di una riga

Ora abbiamo delle leggere differenze su alcuni parametri, analizziamo prima la funzione readData:

- totalRow: variabile utilizzata come risultato che restituisce il numero totale delle righe
- columnNumber: variabile utilizzata come risultato che restituisce il numero totale delle colonne

Nel caso della funzione openAndSample ho:

- sample: indica la grandezza del campione s per Big-Means che vogliamo estrarre dal file
- seed: indica il seme da utilizzare per il generatore di numeri casuali

Attraverso queste due funzioni siamo in grado di acquisire il dataset a seconda dei casi in cui ci troviamo. Ora disponiamo di tutti gli elementi necessari per costruire il nostro algoritmo Big-Means. Lo pseudocodice seguito è quello citato nella sezione precedente 3. Analizziamo l'itestazione della funzione:

```
double bigmeans(string fileName,int sample, int labelColumn,
int headerRow,char delimiter,char lectureType,int k,
int maxIterationBigMeans,
vector<vector<double>>& final_Centroids);
```

Alcuni dei parametri passati a questa funzione sono finalizzati alla lettura del file, analizziamo i parametri differenti:

- k: numero di centroidi desiderati
- maxIterationBigMeans: il numero di iterazioni che Big-Means deve effettuare
- finalCentroids: una variabile utilizzata come risultato che contiene i centroidi finali

Nella prima iterazione di Big-Means scelgo il primo centroide in maniera casuale e successivamente determino gli altri con l'algoritmo K-Means++, in tutte le altre iterazioni controllo la presenza di centroidi degeneri, nel caso positivo li elimino e li rimpiazzo utilizzando K-Means++.

```
if (iteration == 0)
{
    centroids.push_back(chooseFirstCentroid(points));
    tmp_centroids = centroids;
    tmp_centroids = kmeansplusplus(points,k,tmp_centroids);
}

//se ho centroidi degeneri li sistemo con kmeans++
if (tmp_centroids.size() < k)
{
    tmp_centroids = kmeansplusplus(points,k,tmp_centroids);
}
```

Quello che viene fatto successivamente è chiamare la funzione K-Means, questa determinerà i nuovi centroidi sul campione. A questo punto i centroidi verranno mantenuti solo se la funzione obiettivo per Big-Means risulta migliorativa.

```
    float epslon = 0.0001;
    loss = kmeans(points,tmp_centroids,tmp_degenerateCentroids,k,epslon,300);

    //Nella prima iterazione prendo la prima loss che trovo
    if (iteration == 0)
    {
        cout << endl << "BIG MEANS LOSS (iteration "
        << iteration << "): " << fixed << setprecision(0)
        << loss << endl;
        centroids = tmp_degenerateCentroids;
        best_loss = loss;
    }

    if (best_loss > loss)
    {
        cout << endl << "BIG MEANS LOSS (iteration " << iteration
        << "): " << fixed << setprecision(0) << loss << endl;
        cout << "MIGLIORAMENTO RISPETTO ALLA LOSS PRECEDENTE
        (iteration " << iteration << "): " << fixed
        << setprecision(5) << ((best_loss - loss)/best_loss)*100
        << "%" << endl << endl;
        centroids = tmp_degenerateCentroids;
        best_loss = loss;
    }

    iteration++;
```

L'algoritmo si ferma quando viene soddisfatta una condizione di terminazione, che in questo caso è un numero di iterazioni da noi definito.

3.4 Analisi sperimentale

Nell'articolo relativo a Big-Means [6] sono presenti molti dataset utilizzati per il confronto delle soluzioni dell'algoritmo, quello che è stato fatto è confrontatare i risultati forniti da Big-Means con quelli degli algoritmi di clustering che rappresentano attualmente lo stato dell'arte. In questa istanza si sono utilizzati una parte dei dataset per riprodurre i risultati ottenuti dai ricercatori. Introduciamo la sintassi necessaria a comprendere le tabelle di confronto successive:

- k: numero di centroidi

- f_{best}: valore della funzione di costo ottenuta con gli algoritmi attualmente considerati lo stato dell'arte
- f: funzione di costo ottenuta con Big-Means
- $E_A = \frac{f - f_{best}}{f_{best}} \cdot 100\%$: errore normalizzato commesso in percentuale
- n_{exec}: numero di run di Big-Means
- n_s: numero di iterazioni di Big-Means
- s: campione utilizzato a ogni iterazione di Big-Means
- m: numero di punti
- n: dimensionalità dei punti
- *cpu*: tempo medio impegato per l'esecuzione di Big-Means in secondi

Per i dataset di dimensione più piccola è stata utilizzata la metrica di *Silhouette* per valutare la bontà dei cluster. Per ogni punto $i \in C_I$ (ogni punto i nel cluster C_I) definisco:

$$a(i) = \frac{1}{|C_I| - 1} \sum_{j \in C_I, j \neq i} d(i, j) \tag{10}$$

con $|C_I|$ numero di punti nel cluster C_I e con $d(i, j)$ una qualsiasi metrica di distanza. Quindi con $a(i)$ indichiamo la distanza media tra il punto $i - esimo$ e i restanti punti del cluster. A questo punto definiamo per ogni punto $i \in C_I$:

$$b(i) = \min_{J \neq I} \frac{1}{|C_J|} \sum_{j \in C_J} d(i, j) \tag{11}$$

effettuata la media del punto $i - esimo$ rispetto ai punti del cluster $J - esimo$, per ogni cluster diverso da quello a cui il punto i-esimo appartiene, selezioniamo il valore medio più piccolo. A questo punto possiamo definire la metrica di Silhouette come:

$$s(i) = \frac{b(i) - a(i)}{max\{b(i), a(i)\}} \tag{12}$$

Questa metrica è stata applicata solo sui dataset più piccoli poichè l'algoritmo per il calcolo del Silhouette score ha complessità $O(N^2)$ (l'algoritmo è stato implementato in C++ senza l'utilizzo di alcuna libreria). Nelle seguenti tabelle vediamo il confronto dei risultati ottenuti nei nostri esperimenti su differenti dataset:

Dataset: US Census Data 1990; $m = 2458285, n = 68$								
Articolo Big-Means [6]								
k	$f_{\text{best}} \times 10^8$	s	n_{s}	n_{exec}	min E_A	mean E_A	max E_A	*cpu*
2	18.39812	6000	241	20	0.06	0.32	0.92	1.73
3	6.1591	6000	208	20	0.04	24.69	164.26	1.62
5	3.35214	6000	190	20	0.08	5.57	28.24	1.85
10	2.36352	6000	122	20	1.46	6.06	11.98	1.88
15	2.04097	6000	74	20	2.08	5.44	10.26	1.65
20	1.81278	6000	53	20	2.08	4.91	9.03	1.78
25	1.64602	6000	32	20	2.53	4.97	8.17	1.51
Risultati ottenuti								
k	$f_{\text{best}} \times 10^8$	s	n_{s}	n_{exec}	min E_A	mean E_A	max E_A	*cpu*
2	18.39812	6000	200	10	-0.01	0.16	0.28	40.21
3	6.1591	6000	200	10	0.09	16.47	163.58	45.08
5	3.35214	6000	200	10	0.12	9.76	33.53	55.40
10	2.36352	6000	150	10	0.22	7.66	19.35	85.73
15	2.04097	6000	80	10	3.48	8.46	20.82	138.38
20	1.81278	6000	55	10	4.11	6.82	11.81	199.52
25	1.64602	6000	40	10	3.14	9.04	18.28	239.12

Tabella 3.1: Test eseguito sul dataset US Census Data 1990 con $m = 2458285, n = 68$

Dataset: Protein Homology; $m = 145751, n = 74$								
Articolo Big-Means [6]								
k	$f_{\text{best}} \times 10^{11}$	s	n_{s}	n_{exec}	min E_A	mean E_A	max E_A	cpu
2	15.20433	56000	50	15	0.09	1.44	1.89	1.55
3	8.07129	56000	50	15	0.12	0.77	1.76	2.19
5	5.30537	56000	30	15	0.23	0.63	2.0	1.89
10	3.3767	56000	20	15	0.05	1.75	18.99	2.14
15	2.86473	56000	10	15	0.18	0.72	1.37	2.78
20	2.5732	56000	8	15	0.27	1.23	1.82	2.8
25	2.38539	56000	5	15	0.37	1.37	2.36	3.06
Risultati ottenuti								
k	$f_{\text{best}} \times 10^{11}$	s	n_{s}	n_{exec}	min E_A	mean E_A	max E_A	cpu
2	15.20433	56000	41	10	0.24	1.62	1.89	40.24
3	8.07129	56000	50	10	0.01	0.33	1.27	45.08
5	5.30537	56000	30	10	0.08	0.49	1.61	55.40
10	3.3767	56000	13	10	0.14	15.29	19.58	84.65
15	2.86473	56000	7	10	0.06	5.73	26.79	138.38
20	2.5732	56000	3	10	1.03	1.35	2.05	199.52
25	2.38539	56000	3	10	0.35	1.49	2.51	239.12

Tabella 3.2: Test eseguito sul dataset Protein Homology con $m = 145751, n = 74$

Dataset: MFCCs for Speech Emotion Recognition; $m = 85134, n = 58$								
Articolo Big-Means [6]								
k	$f_{\text{best}} \times 10^{9}$	s	n_{s}	n_{exec}	min E_A	mean E_A	max E_A	cpu
2	0.74513	12000	53	20	0.01	0.04	0.07	0.45
3	0.50215	12000	52	20	0.02	0.05	0.13	0.47
5	0.3456	12000	34	20	0.03	0.06	0.12	0.51
10	0.21763	12000	22	20	0.06	1.73	3.4	0.71
15	0.17608	12000	10	20	0.14	1.23	5.24	0.63
20	0.15383	12000	8	20	0.24	1.24	3.64	0.74
25	0.14109	12000	5	20	0.45	1.6	4.71	0.69
Risultati ottenuti								
k	$f_{\text{best}} \times 10^{9}$	s	n_{s}	n_{exec}	min E_A	mean E_A	max E_A	cpu
2	0.74513	12000	50	10	0.03	0.05	0.09	24.53
3	0.50215	12000	50	10	0.07	0.08	0.10	31.19
5	0.3456	12000	50	10	0.07	0.10	0.22	45.64
10	0.21763	12000	40	10	0.16	2.23	3.20	86.71
15	0.17608	12000	30	10	0.24	1.37	3.16	105.11
20	0.15383	12000	20	10	0.71	1.25	2.18	120.56
25	0.14109	12000	10	10	0.66	1.6	6.30	116.26

Tabella 3.3: Test eseguito sul dataset MFCCs for Speech Emotion Recognition con $m = 85134, n = 58$

Dataset: ISOLET; $m = 7797, n = 617$								
Articolo Big-Means [6]								
k	$f_{\text{best}} \times 10^5$	s	n_{s}	n_{exec}	min E_A	mean E_A	max E_A	cpu
2	7.2194	4000	166	15	0.02	0.03	0.05	2.88
3	6.78782	4000	146	15	0.03	0.23	0.62	3.39
5	6.13651	4000	108	15	0.05	0.71	1.76	3.68
10	5.28577	4000	54	15	0.13	0.89	2.16	3.48
15	4.87391	4000	40	15	0.22	1.28	2.59	3.82
20	4.60857	4000	23	15	0.42	1.07	2.82	3.57
25	4.44323	4000	20	15	0.11	0.93	2.51	3.85
Risultati ottenuti								
k	$f_{\text{best}} \times 10^5$	s	n_{s}	n_{exec}	min E_A	mean E_A	max E_A	cpu
2	7.2194	4000	100	10	0.01	0.01	0.02	79.87
3	6.78782	4000	100	10	0.02	0.26	0.60	104.85
5	6.13651	4000	100	10	0.05	0.48	1.73	156.91
10	5.28577	4000	50	10	0.31	1.14	1.67	143.94
15	4.87391	4000	40	10	0.73	2.00	4.26	193.03
20	4.60857	4000	30	10	0.28	1.14	2.86	248.62
25	4.44323	4000	20	10	0.11	1.16	2.59	281.36

Tabella 3.4: Test eseguito sul dataset ISOLET con $m = 7797, n = 617$

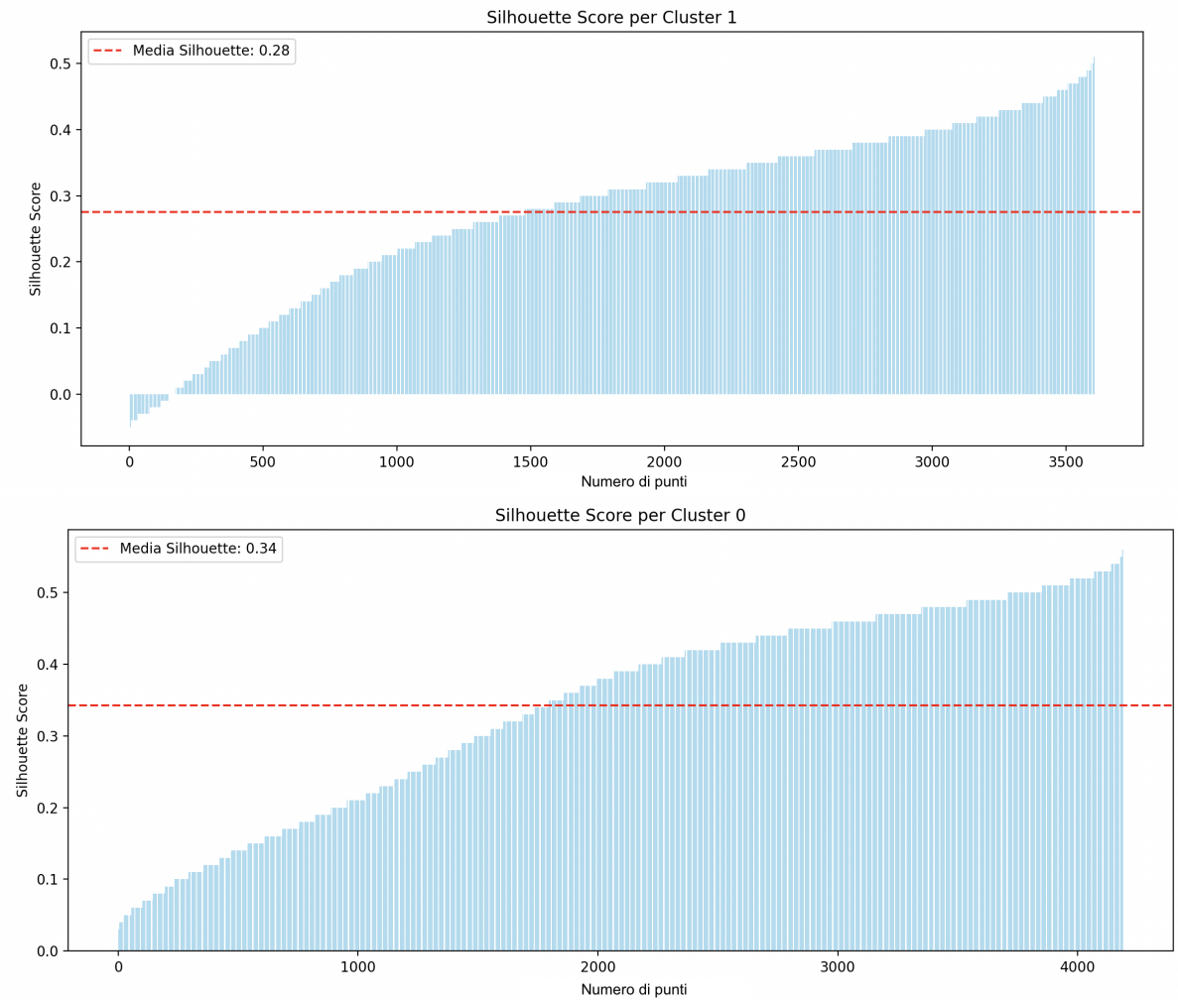

Figura 3.1: Silhouette score sul dataset isolet considerando $k = 2$ e $s = 4000$

Dataset: Sensorless Drive Diagnosis; $m = 58509, n = 48$								
Articolo Big-Means [6]								
k	$f_{\text{best}} \times 10^7$	s	n_{s}	n_{exec}	min E_A	mean E_A	max E_A	*cpu*
2	3.88116	58508	24	40	-0.0	10.02	100.19	0.57
3	2.91313	58508	20	40	-0.0	6.4	16.97	0.56
5	1.93651	58508	19	40	0.0	4.44	37.85	0.79
10	0.98472	58508	9	40	-2.42	4.81	23.18	0.88
15	0.62816	58508	3	40	0.01	1.43	23.44	1.07
20	0.49884	58508	1	40	-0.6	1.95	5.81	1.22
25	0.42225	58508	1	40	0.45	3.03	9.28	1.52
Risultati ottenuti								
k	$f_{\text{best}} \times 10^7$	s	n_{s}	n_{exec}	min E_A	mean E_A	max E_A	*cpu*
2	3.88116	30000	50	5	2.49	62.77	100.27	33.75
3	2.91313	30000	45	5	14.19	60.10	155.58	44.93
5	1.93651	30000	40	5	42.91	132.33	264.49	76.95
10	0.98472	30000	35	5	119.45	244.5	329.16	163.26
15	0.62816	30000	30	5	36.77	313.03	690.77	244.15
20	0.49884	30000	20	5	44.71	276.23	503.69	256.29
25	0.42225	30000	10	5	114.41	177.36	260.08	248.11

Tabella 3.5: Test eseguito sul dataset Sensorless Drive Diagnosis con $m = 58509, n = 48$

Dataset: Online News Popularity; $m = 39644, n = 58$								
Articolo Big-Means [6]								
k	$f_{\text{best}} \times 10^{14}$	s	n_{s}	n_{exec}	min E_A	mean E_A	max E_A	*cpu*
2	9.53913	10000	87	20	0.0	0.01	0.04	0.4
3	5.91077	10000	53	20	0.01	4.85	24.1	0.43
5	3.09885	10000	48	20	0.04	3.73	31.06	0.47
10	1.17247	10000	16	20	0.29	3.97	17.66	0.39
15	0.77637	10000	9	20	0.81	5.12	16.32	0.49
20	0.59809	10000	6	20	2.68	5.96	10.62	0.44
25	0.49616	10000	4	20	3.07	6.24	10.59	0.46
Risultati ottenuti								
k	$f_{\text{best}} \times 10^{14}$	s	n_{s}	n_{exec}	min E_A	mean E_A	max E_A	*cpu*
2	9.53913	10000	80	5	0.57	0.58	0.59	22.50
3	5.91077	10000	70	5	0.95	0.97	1.0	32.87
5	3.09885	10000	60	5	1.75	1.79	1.82	40.79
10	1.17247	10000	50	5	5.97	19.61	58.28	59.63
15	0.77637	10000	40	5	24.12	25.83	30.59	97.45
20	0.59809	10000	30	5	29.20	35.27	41.31	132.96
25	0.49616	10000	20	5	39.44	40.85	42.53	149.73

Tabella 3.6: Test eseguito sul dataset Online News Popularity con $m = 39644, n = 58$

Dataset: Gas Sensor Array Drift; $m = 13910, n = 128$								
Articolo Big-Means [6]								
k	$f_{\text{best}} \times 10^{13}$	s	n_{s}	n_{exec}	min E_A	mean E_A	max E_A	*cpu*
2	7.91186	9000	471	30	0.07	0.17	0.46	4.61
3	5.02412	9000	388	30	0.07	0.2	0.45	3.93
5	3.22394	9000	238	30	0.03	4.59	8.4	3.73
10	1.65524	9000	114	30	-0.11	3.91	22.65	4.24
15	1.13801	9000	74	30	-0.74	5.38	13.99	3.79
20	0.87916	9000	67	30	0.12	4.0	9.99	4.41
25	0.72274	9000	52	30	0.42	4.32	12.46	4.52
Risultati ottenuti								
k	$f_{\text{best}} \times 10^{13}$	s	n_{s}	n_{exec}	min E_A	mean E_A	max E_A	*cpu*
2	7.91186	9000	200	5	0.02	0.07	0.13	91.36
3	5.02412	9000	150	5	0.06	0.09	0.14	89.44
5	3.22394	9000	100	5	6.90	7.75	8.35	98.69
10	1.65524	9000	80	5	7.70	12.62	15.35	185.57
15	1.13801	9000	60	5	20.07	24.57	27.58	226.71
20	0.87916	9000	50	5	13.85	24.25	40.72	275.68
25	0.72274	9000	40	5	17.15	27.35	46.00	337.48

Tabella 3.7: Test eseguito sul dataset Gas Sensor Array Drift con $m = 13910, n = 128$

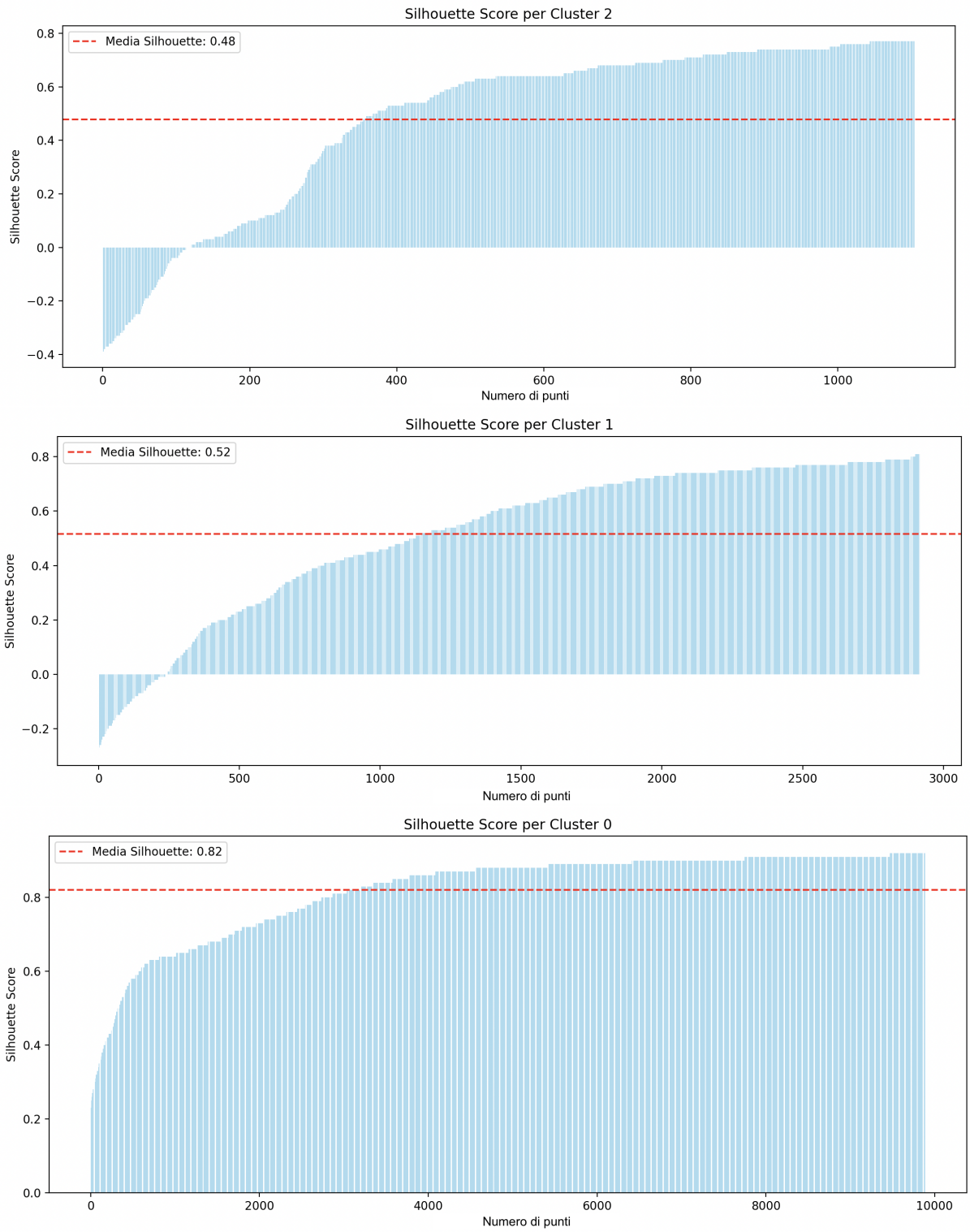

Figura 3.2: Silhouette score sul dataset Gas Sensor Array Drift considerando $k = 3$ e $s = 9000$

Dataset: 3D Road Network; $m = 434874, n = 3$								
Articolo Big-Means [6]								
k	$f_{\text{best}} \times 10^6$	s	n_{s}	n_{exec}	min E_A	mean E_A	max E_A	*cpu*
2	49.13298	100000	17	40	0.0	0.01	0.03	0.28
3	22.77818	100000	15	40	0.0	0.01	0.09	0.29
5	8.82574	100000	12	40	0.0	0.03	0.12	0.3
10	2.56661	100000	6	40	0.01	0.22	1.01	0.38
15	1.27069	100000	3	40	0.05	0.48	1.51	0.44
20	0.80865	100000	2	40	0.06	1.13	3.18	0.47
25	0.59259	100000	2	40	0.33	1.72	4.89	0.6
Risultati ottenuti								
k	$f_{\text{best}} \times 10^6$	s	n_{s}	n_{exec}	min E_A	mean E_A	max E_A	*cpu*
2	49.13298	100000	20	5	-0.01	-0.00	0.01	15.97
3	22.77818	100000	20	5	-0.01	0.00	0.01	20.85
5	8.82574	100000	15	5	0.00	0.25	0.03	26.38
10	2.56661	100000	15	5	0.05	0.07	0.12	68.31
15	1.27069	100000	10	5	0.06	0.48	1.02	191.19
20	0.80865	100000	10	5	0.06	0.59	1.80	142.58
25	0.59259	100000	5	5	0.46	1.39	2.27	452.71

Tabella 3.8: Test eseguito sul dataset 3D Road Network con $m = 434874, n = 3$

Dataset: Pla85900; $m = 85900, n = 2$								
Articolo Big-Means [6]								
k	$f_{\text{best}} \times 10^{15}$	s	n_{s}	n_{exec}	min E_A	mean E_A	max E_A	*cpu*
2	3.74908	14000	215	40	0.00	0.74	1.48	0.55
3	2.28057	14000	156	40	0.00	0.04	0.14	0.48
5	1.33972	14000	243	40	0.02	0.51	2.88	0.64
10	0.68294	14000	151	40	0.06	0.41	1.18	0.52
15	0.46029	14000	107	40	0.1	0.55	2.56	0.51
20	0.34988	14000	82	40	0.17	0.66	1.62	0.53
25	0.28259	14000	68	40	0.15	0.92	1.68	0.58
Risultati ottenuti								
k	$f_{\text{best}} \times 10^{15}$	s	n_{s}	n_{exec}	min E_A	mean E_A	max E_A	*cpu*
2	3.74908	14000	200	5	00.1	0.87	1.48	22.40
3	2.28057	14000	200	5	0.00	0.01	0.04	26.44
5	1.33972	14000	100	5	0.03	0.96	2.94	23.19
10	0.68294	14000	100	5	0.47	0.60	0.71	66.30
15	0.46029	14000	100	5	0.14	0.36	1.05	120.97
20	0.34988	14000	100	5	0.22	0.48	0.96	176.63
25	0.28259	14000	100	5	0.45	0.80	1.13	217.19

Tabella 3.9: Test eseguito sul dataset Pla85900 con $m = 85900, n = 2$

Dataset: D15112; $m = 15112, n = 2$								
Articolo Big-Means [6]								
k	$f_{\text{best}} \times 10^{11}$	s	n_{s}	n_{exec}	min E_A	mean E_A	max E_A	cpu
2	3.68403	8000	976	15	0.0	0.02	0.06	1.18
3	2.5324	8000	637	15	0.0	0.04	0.09	0.88
5	1.32707	8000	684	15	0.01	2.26	16.73	0.76
10	0.64491	8000	741	15	0.08	1.33	3.92	1.27
15	0.43136	8000	459	15	0.14	1.25	7.44	0.94
20	0.32177	8000	542	15	0.36	1.25	2.7	1.4
25	0.25308	8000	379	15	0.27	0.84	1.49	1.26
Risultati ottenuti								
k	$f_{\text{best}} \times 10^{11}$	s	n_{s}	n_{exec}	min E_A	mean E_A	max E_A	cpu
2	3.68403	8000	900	5	-0.00	0.01	0.03	44.32
3	2.5324	8000	800	5	0.00	0.02	0.04	58.56
5	1.32707	8000	700	5	0.00	3.36	16.70	68.99
10	0.64491	8000	600	5	0.06	1.38	3.75	153.60
15	0.43136	8000	500	5	0.29	1.19	2.40	234.08
20	0.32177	8000	500	5	1.17	1.57	2.06	341.50
25	0.25308	8000	500	5	0.25	0.51	0.89	467.07

Tabella 3.10: Test eseguito sul dataset D15112 con $m = 15112, n = 2$

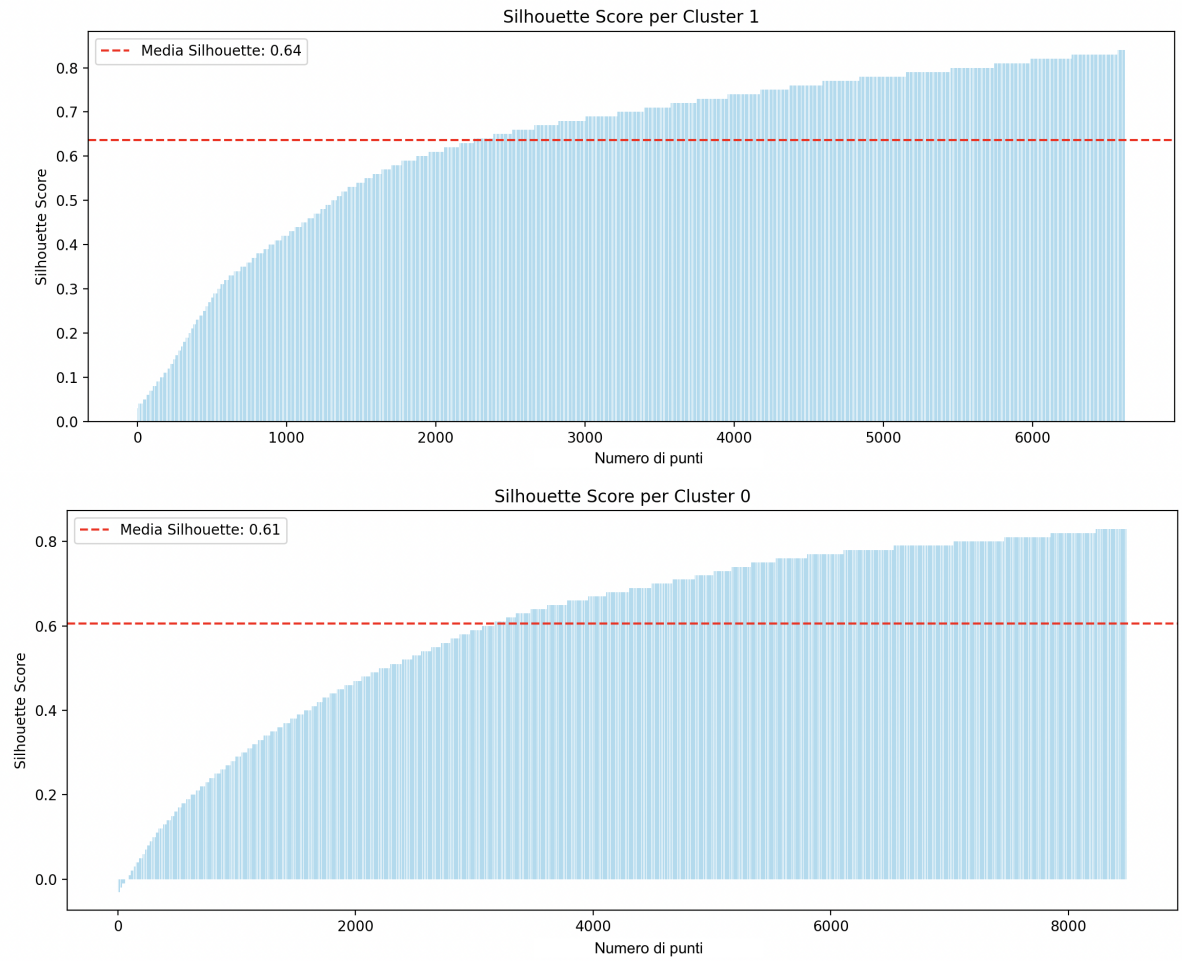

Figura 3.3: Silhouette score sul dataset D15112 considerando $k = 2$ e $s = 8000$

4 Conclusioni

L'algoritmo è stato implementato nella sua versione sequenziale, quindi è possibile notare una forte discrepanza tra i tempi di esecuzione dei ricercatori e quelli del nostro esperimento; nella versione ufficiale dell'algoritmo e nei risultati riportati per i benchmark i ricercatori hanno utilizzato Big-Means, KMeans++ e KMeans nelle loro versioni parallele. Attraverso gli esperimenti effettuati riproducendo l'algoritmo Big-Means, quello che è stato possibile notare è che generalmente, trovando un compromesso sufficientemente stabile tra il valore dei campioni s e il numero di centroidi k è possibile ottenere delle soluzioni prossime alle migliori soluzioni presenti nello stato dell'arte e, a volte, anche migliorative. D'altro canto dosare questi due parametri in maniera corretta non risulta affatto semplice, oltretutto, alcuni dataset non presentano buoni risultati con campionamenti di taglia inferiore all'itero dataset. Vediamo quanto appena detto nel dataset 3.5, si può osservare che i ricercatori ottengono delle soluzioni migliorative utilizzando come campione l'intero dataset, quindi all'atto pratico quello che si fa è iterare n_s volte l'algoritmo K-Means su tutti i punti. Nella nostra riproduzione abbiamo utilizzato un campione di dati inferiore, che, come si può vedere, porta a dei risultati nettamente differenti da quelli desiderati. Dato il contributo random dell'algoritmo, in alcuni dataset siamo riusciti ad ottenere delle soluzioni migliorative rispetto allo stato dell'arte 3.8, questo ci dimostra che, nelle condizioni corrette, l'algoritmo performa in modo ottimo, anche se, in applicazioni reali in cui non è possibile ottenere delle metriche per l'intero dataset, la soluzione fornita da Big-Means (e quindi i centroidi "migliori" trovati dall'algoritmo) è accettata come "un atto di fede", dato che non siamo in grado realmente di stabilire le performance rispetto all'analisi dell'intero dataset. Possiamo concludere affermando che l'algoritmo presenta ottime prestazioni a patto di effettuare un "tuning" corretto dei parametri s e k, e che queste ottime prestazioni sono però isolate, ovviamente, a dataset che si prestano a una risoluzione di tipo point assignment, a discapito di questi problemi l'algoritmo si presta a risolvere dei problemi di clustering su grandi quantità di dati, quindi a patto di perdere in qualità per quanto riguarda i centroidi, si guadagna in tempi di esecuzione in modo non trascurabile.

A Esecuzione dell'algoritmo

Illustriamo i passaggi da eseguire per fare un'esecuzione dell'algoritmo:

- Entrare da terminale nella cartella "code" e lanciare il comando `make` per compilare i file C++.
- Spostarsi, da terminale, all'interno della cartella `bin`

A questo punto bisognerà lanciare il seguente comando:

Listing 1: Lista dei parametri da utilizzare per l'esecuzione dell'algoritmo

```
./main -f <file_dataset> -s <grandezza_del_campione> -r <riga_iniziale>
-c <colonna_da_eliminare> -d <delimitatore> -t <tipo_di_lettura>
-k <centroidi> -i <iterazioni_bigMeans> -L
```

Commentiamo quanto appena detto:

- -f: inserire il nome del file con eventuale path se necessario
- -s: indicare con un numero intero la grandezza del campione di Big-Means
- -r: indicare con un numero intero la riga da cui si vuole iniziare a leggere (se volessi leggere dall'inizio il file inserirei il numero 1)
- -c: indicare con un numero intero la colonna che si vuole eliminare; se non si vuole eliminare nessuna colonna inserire il valore -1
- -d: ogni riga nel file rappresenta un punto, le varie coordinate del punto sono separate da un carattere, indicare tra le virgolette il carattere separatore
- -t: esistono due tipi di lettura, quella basata su un campionamento random dei punti direttamente dal file, che indichiamo con il comando 'l', e quella relativa alla lettura totale del file che indichiamo con 'f'
- -k: indicare con un numero intero i centroidi desiderati
- -i: indicare con un numero intero il numero di iterazioni di Big-Means che si vogliono effettuare
- -L: flag da inserire se, finito l'algoritmo Big-Means, si vogliono utilizzare i centroidi per calcolare la funzione di loss rispetto a tutto il dataset (flag da utilizzare in caso di benchmark o dataset ridotti)

Vediamo un esempio di esecuzione dell'algoritmo sul dataset "Sensorless Drive Diagnosis":

Listing 2: Un esempio di esecuzione dell'algoritmo sul dataset Sensorless Drive Diagnosis con i settings per il caso $k = 2$.

```
./main -f "../../datasets/Sensorless_drive_diagnosis.txt" -s 58508 -r 1
-c 49 -d ' ' -t f -k 2 -i 10 -L
```

Per visualizzare tutti i flag e tutte le informazioni disponibili riguardo i parametri da passare prima dell'esecuzione del programma, basta digitare:

```
./main -h
```

e questo ci restituirà tutte le informazioni a riguardo in maniera più dettagliata.

B Struttura del dataset

Come detto anche in precedenza l'algoritmo accetta dei dataset di tipo numerico, quindi senza variabili di tipo categorico. Quello che si può fare nel caso di colonne che non rappresentano informazioni utili al clustering, o di feature di tipo categorico che vanno rimosse, è ignorare una colonna inserendo il numero relativo alla sua posizione nel comando di lancio del programma. Molti dataset presenti nell'artico originale di Big-Means [6] non sono direttamente utilizzabili, questo perchè sono suddivisi in differenti file. È stata nostra premura aggregare i molteplici file (se necessario) in un unico file in modo da poter lavorare in maniera più agevole. Lo stesso ragionamento vale per i separatori, alcuni file presentavano dei separatori alquanto scomodi, questi sono stati manualmente modificati attraverso degli editor di testo in spazi o

virgole, che sono da preferirsi per una questione di interoperabilità. Altrettanti dataset avevano una serie di colonne non utili al clustering (poichè utilizzati nella classificazione), dato che l'algoritmo è in grado di non considerare una colonna, quello che è stato fatto è agire manualmente attraverso alcuni programmi python per rimuovere queste colonne non utili al nostro scopo. In definitiva, è stata apportata una lieve modifica strutturale ad alcuni dataset cosicché si possa effettuare il clustering in modo corretto rispettando quanto fatto dagli autori. Alcuni dataset forniti nell'articolo presentano dei problemi per cui non è stato possibile effettuare dei test, in particolare Music Analysis. Per questioni relative all'attrezzatura a disposizione per il test dell'algoritmo, non è stato possibile effettuare il calcolo di E_A per il dataset Cord19, date le sue grandi dimensioni, ciò implica che l'algoritmo Big-Means funziona su questo dataset ma non si è in grado di calcolare la loss rispetto a tutti i punti per poter effettuare un confronto con lo stato dell'arte.

C Benchmark

Per avviare la modalità benchmark del programma necessitiamo di aggiungere alcune informazioni e alcuni flag al momento del lancio. Vediamo il comando relativo al lancio del benchmark:

Listing 3: Lista dei parametri da utilizzare per l'esecuzione del benchmark dell'algoritmo

```
./main -f <file_dataset> -s <grandezza_del_campione> -r <riga_iniziale>
-c <colonna_da_eliminare> -d <delimitatore> -t <tipo_di_lettura>
-k <centroidi> -i <iterazioni_bigMeans>
-B -S -F <file_di_output> -E <f_best>
```

Vediamo nel dettaglio i parametri aggiuntivi per il benchmark ed il loro funzionamento:

- -B: aggiungendo questo flag si attiva la modalità banchmark
- -S: aggiungendo questo flag e tenendo attivo il flag -B possiamo calcolare anche il Silhouette Score
- -F: richiede il nome del file di output in cui verranno inseriti i risultati del benchmark
- -E: richiede il valore della f_{best} per quel test che stiamo effettuando

Quando si attiva il flag -B non necessitiamo di inserire -L poichè implicito nel benchmark. Vediamo un esempio:

Listing 4: Un esempio di esecuzione di benchmark dell'algoritmo sul dataset Sensorless Drive Diagnosis con i settings per il caso $k = 2$.

```
./main -f "../../datasets/Sensorless_drive_diagnosis.txt" -s 58508 -r 1
-c 49 -d '-' -t f -k 2 -i 10 -B -F "sensorless.txt" -E 38811600
```

Il file di output del benchmark verrà automaticamente creato e inserito nella cartella benchmark, quindi non necessitiamo di crearlo anticipatamente. Nella cartella *benchmark* è presente un file python che permette l'esecuzione dei benchmark in diverse configurazioni. Il file si presenta in questo modo:

```
import os

file_path = "3dSpatialNetwork.txt"
dataset = "../../datasets/3D_spatial_network.txt"
main_path = "../bin/main"
iteration_for_setting = 5

header = "Grandezza campione,Numero di centroidi,Iterazioni di Big Means,
Tempo trascorso,LOSS FINALE,Ea\n"

with open(file_path, "a") as file:
    file.write(header)

parameters = [
    (100000, 1, 1, ',', 'f', 2, 5, 49132980),
    (100000, 1, 1, ',', 'f', 3, 5, 22778180),
    (100000, 1, 1, ',', 'f', 5, 5, 8825740),
    (100000, 1, 1, ',', 'f', 10, 5, 2566610),
```

```
    (100000, 1, 1, ',', 'f', 15, 5, 1270690),
    (100000, 1, 1, ',', 'f', 20, 5, 808650),
    (100000, 1, 1, ',', 'f', 25, 5, 592590),
]

for param in parameters:
    command = (
        f"{main_path} -f \"{dataset}\" "
        f"-s {param[0]} -r {param[1]} -c {param[2]} -d '{param[3]}' -t '{param[4]}' "
        f"-k {param[5]} -i {param[6]} -E {param[7]} -F \"{file_path}\" -B"
    )

    for _ in range(iteration_for_setting):
        os.system(command)

    file.write(header)
```

Commentiamo alcune variabili principali di questo codice che vanno modificate a seconda dei benchmark che si vuole eseguire:

- file_path: rappresenta il nome del file in cui finiranno i benchmark
- dataset: il nome del file contenente il dataset (ed eventuale path)
- iteration_for_setting: rappresenta il numero di esecuzioni dell'algoritmo che vengono effettuate considerando un settaggio dei parametri
- parameters: sono i differenti parametri che si vogliono dare all'algoritmo per effettuare i benchmark

Questi valori appena indicati vanno modificati in base alle nostre esigenze, nel codice dell'esempio appena fornito verranno effettuate 5 iterazioni su ogni configurazione fornita, in questo caso abbiamo 7 configurazioni, quindi in totale verranno effettuate 70 esecuzioni. Questo script python per essere eseguito necessita unicamente del comando da terminale:

```
python3 benchmark.py
```

5 Riferimenti Bibliografici

[1] P. S. Bradley, U. Fayyad e C. Reina, ≪Scaling Clustering Algorithms to Large Databases,≫ in *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, ser. KDD'98, New York, NY: AAAI Press, 1998, pp. 9–15.

[2] S. Lloyd, ≪Least squares quantization in PCM,≫ 2, vol. 28, 1982, pp. 129–137. DOI: 10.1109/TIT.1982.1056489.

[3] D. Arthur e S. Vassilvitskii, ≪K-Means++: The Advantages of Careful Seeding,≫ in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '07, New Orleans, Louisiana: Society for Industrial e Applied Mathematics, 2007, pp. 1027–1035, ISBN: 9780898716245.

[4] S. Yang, D. Towey e Z. Q. Zhou, ≪Metamorphic Exploration of an Unsupervised Clustering Program,≫ mag. 2019, pp. 48–54. DOI: 10.1109/MET.2019.00015.

[5] ≪ML — K-means++ Algorithm - GeeksforGeeks — geeksforgeeks.org.≫

[6] R. Mussabayev, N. Mladenovic, B. Jarboui e R. Mussabayev, ≪How to Use K-means for Big Data Clustering?,≫ vol. 137, 2023, p. 109 269. DOI: https://doi.org/10.1016/j.patcog.2022.109269. indirizzo: https://www.sciencedirect.com/science/article/pii/S0031320322007488.

Elenco delle figure