Implementazione dell’algoritmo DBSCAN e sua ottimizzazione mediante istruzioni SIMD

Indice

I. Prefazione

Nella seguente relazione verranno ignorate le caratteristiche dell'algoritmo in oggetto, per concentrarsi di più sulle scelte implementative che hanno portato ad una ottimizzazione delle prestazioni dello stesso. Inoltre, lo sviluppo della versione a 64 bit non si discosta di molto da quello della versione a 32 bit; dunque, l'intera trattazione verterà sulle scelte progettuali prese durante lo sviluppo dell'artefatto a 32 bit, evidenziando, dove necessario, le differenze con l'artefatto a 64 bit. In fine sarà dedicato un paragrafo alle scelte e considerazioni inerenti all'utilizzo del framework OpenMP.

È, inoltre, presente un'appendice in cui si discutono le modifiche apportate al template fornito.

II. Implementazione dell'algoritmo

L'algoritmo è stato implementato seguendo lo schema descritto in [1], tranne che per un piccolo particolare, ovvero la funzione *regionQuery* [1] restituisce un insieme di punti che non contiene il punto oggetto dell'interrogazione; questa scelta, come le altre, è stata presa per ottimizzare le prestazioni, in particolare così facendo non è richiesto che venga eliminato, successivamente, il punto dall'insieme di punti. Per lo stesso motivo, l'equivalente della funzione *first* [1], estrae un elemento dall'insieme in modo permanente, ciò per evitare, in seguito, l'eliminazione dell'elemento dall'insieme stesso. Inoltre, l'equivalente della funzione *first* [1] in realtà restituisce l'ultimo elemento dell'insieme; questa scelta è stata resa necessaria per una gestione ottimale della struttura dati Punti che discuteremo in seguito.

In conclusione, le scelte sopra descritte hanno portato ad una differente gestione degli indici e delle condizioni nell'algoritmo, rispetto a quanto descritto in [1]; per il resto, i due algoritmi sono praticamente identici.

III. Strutture dati

L'implementazione dell'algoritmo, come è prevedibile che sia, ha richiesto l'utilizzo di strutture dati dedicate. Non ci addentreremo nei dettagli strettamente legati alla gestione del codice di queste strutture dati, perché ciò è descritto in modo più che esaustivo all'interno del codice, tramite i commenti di accompagno.

La principale struttura dati è la matrice *ds*, quest'ultima è implementata come un vettore monodimensionale in row-major order; essa, memorizza in ogni riga un punto del dataset, e nella i-esima colonna l'i-esima componente dimensionale del punto stesso. Un'altra struttura dati che ha semplificato l'implementazione dell'algoritmo è la struttura dati *Punti*, avente la seguente struttura:

```c
/*
L'array di interi contiene gli indici delle colonne di "ds", dunque contiene i punti.
Size è la numerosità dell'array.
*/
typedef struct {
    int* indici;  // Puntatore al vettore di interi
    int size;     // Dimensione del vettore
    int capacity; //serve per tenere traccia di quanti spazi di memoria disponiamo
} Punti;
```

Accenniamo al fatto che l'attributo *capacity* è essenziale per la gestione della Garbage Collection di cui parleremo in un paragrafo dedicato. Ovviamente di tale struttura dati è stata realizzata una funzione "costruttore" ed una procedura "distruttore", oltre che i vari metodi di utilità. A livello astratto tutte le variabili di tipo *Punto* vanno intese come un array dinamico, in particolare con una struttura a pila in cui è possibile inserire ed estrarre un elemento in coda, oltre che a sapere la numerosità degli elementi in essa contenuti. L'ultima struttura dati, forse la più importante, utilizzata è un Ball Tree. Esso è un albero binario che supporta in modo efficiente le ricerche di vicinato, ovvero le *regionQuery*; la struttura dei nodi del Ball Tree è la seguente:

```c
#define LEAFSIZE 300    //per capire cos'è bisogna leggere la funzione calcolo_partizione
/*
Un Ball-Tree è una struttura dati utilizzata per la ricerca efficiente di vicini
più prossimi in spazi metrici ad alta dimensione. La partizione dei punti in due
sotto cerchi è un passo chiave nella costruzione di un Ball-Tree.

Un nodo di un BallTree rappresenta una iper-sfera, dunque è caratterizzato da un
centro (un punto di più dimensioni) ed un raggio. Questi nodi costruiscono un
albero binario. I nodi foglia rappresentano delle iper-sfere che contengono i
dei punti, e non altre iper-sfere, questo perchè il numero di punti è troppo
piccolo per determinare una loro partizione in due sotto iper-sfere.
*/
typedef struct BallTreeNode {
    type* centro;                // Centro della ball
    type raggio;                 // Raggio della ball
    struct BallTreeNode *left;   // Sotto-albero sinistro
    struct BallTreeNode *right;  // Sotto-albero destro
    bool isLeaf;
    Punti* punti;                // Punti in questa ball (solo foglie)
} BallTreeNode;
```

Un Ball Tree è un albero binario che si costruisce in modo abbastanza particolare, per semplificarne la spiegazione immaginiamo che il numero di dimensioni coinvolte sia pari a 2 (dunque immaginiamo di muoverci all'interno del piano cartesiano). Inizialmente si ha un insieme di punti contenete tutti i punti del dataset; è facile capire che è possibile definire una circonferenza che contenga tutti questi punti, centrata nel centro di massa dei punti stessi, e con un raggio pari alla distanza maggiore tra il centro ed i punti. Ora, possiamo partizionare questo insieme di punti in due sottoinsiemi, ognuno dei quali è rappresentato da una circonferenza (costruita con gli stessi criteri della precedente). Dunque, si itera questo procedimento finché non si ottiene una circonferenza che contiene al più un numero prefissato di punti (*LEAFSIZE* nel codice). L'utilità di questa struttura dati è dettata dal criterio di partizionamento di un insieme di punti in due sottoinsiemi. Il criterio da noi utilizzato (algoritmicamente descritto all'interno del file contenente il codice C, anche mediante commenti) garantisce che i nodi foglia dell'albero contengano elementi molto vicini tra di loro, in cui la vicinanza è intesa in termini di densità; dunque, è immediato capire che questa struttura dati ottimizza di molto le prestazioni di un algoritmo di clustering che si basa sul concetto di densità, proprio come DBSCAN. In fase di ricerca degli elementi più prossimi ad un punto dato, l'interazione con questa struttura dati avviene come segue: dato un punto *p* ed una sensibilità *eps* si calcola la circonferenza di centro *p* e raggio *eps*, chiameremo questa

circonferenza *C*; dunque, iterativamente a partire dalla radice dell'albero, per ogni nodo, ci si chiede se tale circonferenza ha intersezione non nulla con la circonferenza del nodo. Se la risposta è "sì" allora ci si domanda se il nodo in cui ci troviamo è un nodo foglia o meno. Se il nodo è un nodo foglia, allora si estraggono i punti appartenenti all'intersezione tra la circonferenza *C* e la circonferenza del nodo, altrimenti si itera il procedimento sui nodi figli. Se, invece, la risposta è "no", ovvero la circonferenza *C* ha intersezione nulla con la circonferenza del nodo, allora si interrompe il procedimento per quel dato branch di ricorsione.

In conclusione, è immediato capire che la struttura dati che dovrebbe ridurre di molto i tempi di calcolo è il Ball Tree, infatti quest'ultima, nella migliore delle ipotesi, garantisce una ricerca di costo logaritmico, abbassando di molto il costo quadratico che si ha implementando l'algoritmo DBSCAN tramite una ricerca di vicinato esaustiva. Nonostante tutto, non si notano miglioramenti significativi tra le due implementazioni (all'interno del file contenente il codice C è presente anche una funzione di *regionQuery* basata sulla ricerca esaustiva). Questo fenomeno non ci stupisce molto per svariati motivi:

- costruire il Ball Tree ha un costo significativo e tale costo viene conteggiato nel calcolo del tempo impiegato dal programma per eseguire l'algoritmo;
- l'efficienza del Ball Tree dipende molto dal parametro *LEAFSIZE*, ovvero dalla capacità dei nodi foglia. Intuitivamente, per $LEAFSIZE \to \infty$, la ricerca si comporta come una ricerca esaustiva. Dunque, possiamo dire che certamente esisterà un valore di *LEAFSIZE*, dipendente dal numero di punti del dataset, che consentirà di avere le prestazioni ottimali, ma tale valore non è conosciuto e non sappiamo come calcolarlo;
- il Ball Tree non è un albero fortemente bilanciato (come può esserlo un B-Tree, un B^+-Tree, un R-Tree o un R^*-Tree), dunque può tranquillamente divergere in una lista concatenata, offrendo, appunto, prestazioni pari a quelle di una ricerca esaustiva;
- il meccanismo di Garbage Collection manualmente implementato è intensivamente utilizzato nella costruzione del Ball Tree, andando a produrre un tempo di overhead.

In conclusione, possiamo concludere che l'approccio migliore sarebbe quello di usare diverse strutture dati in funzione dell'input, così come è fatto nel framework sklearn, disponibile in

python. Per motivi di tempo non è stato possibile rielaborare un tale approccio anche in questo progetto. Inoltre, inizialmente si era pensato di utilizzare un R-Tree, così come suggerito in [1], ma ovviamente una tale struttura, che richiede il mantenimento di un forte bilanciamento, avrebbe richiesto molto più tempo per essere implementata. Nonostante tutto è stata sviluppata una bozza della stessa, a cui mancano alcune funzionalità.

IV. Convenzioni utilizzate

Lo scopo finale dell'artefatto di programmazione sviluppato è quello di produrre un vettore contenente un'etichetta per ogni punto del dataset. Ogni etichetta rappresenta un numero intero positivo; se nella posizione i-esima del vettore soluzione troviamo il numero *k*, allora vorrà dire che il punto i-esimo apparterrà al cluster con id *k*.

Dunque, per implementare ciò, si è dovuto decidere come identificare un punto che non appartiene a nessun cluster (ovvero come identificare un rumore) e come identificare un punto che ancora non è stato classificato. Per i primi si è scelto di utilizzare il valore *-1*, per i secondi il valore *-2*. Ciò spiega la presenza della funzione *riempi_out_con_meno_due* quale funzione di preparazione all'avvio dell'algoritmo di clustering vero e proprio. Tale convenzione è stata scelta perché in linea con la convenzione applicata in sklearn. Dunque, il primo cluster avrà id pari a *0*.

Ovviamente, *riempi_out_con_meno_due* non è l'unica funzione di preparazione all'avvio dell'algoritmo di clustering, infatti le due funzioni che si occupano di creare il Ball Tree e di popolarlo rientrano in questa categoria.

A. Distanza euclidea

Un'altra scelta fondamentale è quella della distanza da utilizzare. Avendo a che fare con dei punti multidimensionali, abbiamo a che fare, a tutti gli effetti, con uno spazio vettoriale. Quest'ultimo necessità di una norma, così da indurre una distanza e dunque poter lavorare su uno spazio vettoriale normato. Si è scelto, dunque, di applicare la norma due, ovvero la norma euclidea, così da poter lavorare con la distanza euclidea. Questa scelta, in linea con le convenzioni di sklearn, è sembrata la più appropriata avendo a che fare prima con il concetto di densità e poi con i Ball Tree. Ricordiamo che, essendo lo spazio vettoriale dei

punti riducibile allo spazio vettoriale dei più semplici vettori, lo spazio vettoriale dei punti è finitamente generato; dunque, vi è un'equivalenza asintotica tra le norme scelte e dunque, a livello formale, non cambia nulla se si sceglie una norma rispetto che un'altra. Nonostante ciò, la scelta di una norma diversa (per esempio la norma uno, che avrebbe indotto la distanza di Manhattan) avrebbe comportato il riconoscimento di densità diverse, dunque avrebbe alterato il risultato a livello numerico. Perciò, si è scelto di rimanere in linea con le convenzioni di sklearn così da poter sempre confrontare i risultati del nostro artefatto con quelli proposti dal framework sklearn.

È doveroso osservare che la scelta della norma uno avrebbe ridotto i tempi di calcolo.

V. Garbage Collection

Durante i test effettuati sulle soluzioni parziali, si è riscontrato un problema relativo all'utilizzo della memoria dati. Infatti, quest'ultima era facilmente saturabile per via dell'ingente uso della struttura *Punti*. Per ovviare a ciò si è ricorso all'implementazione di un sistema di Garbage collection. Questo sistema prevede la deallocazione di ogni singola struttura *Punti* dopo il suo utilizzo ultimo; dunque, si deallocano le variabili di tipo *Punti* nei seguenti casi:

- in seguito all'utilizzo ultimo dei risultati delle *regionQuery*;
- in seguito al calcolo di una partizione di un nodo del Ball Tree (infatti solo le foglie contengono i puntatori ai nodi);

Inoltre, il sistema di garbage collection implementato prevede una gestione intelligente dello spazio allocato per ogni singola struttura *Punti*; in particolare, quando si inizializza una variabile di tipo *Punti* si alloca una quantità di memoria prefissata e, man mano che questa area di memoria si satura, si raddoppia e si rialloca tutto. Ovviamente quest'ultima è una pratica costosa (in termini computazionali) ma ci permette di utilizzare il nostro artefatto su dataset di grandi dimensioni.

VI. Ottimizzazione mediante istruzioni SIMD

Per ottimizzare ulteriormente le prestazioni del programma si è cercato di tradurre in codice assembly alcune funzioni, così da poter applicare le istruzioni Single Instruction Multiple Data.

Purtroppo, un'ottimizzazione di questo tipo è utile solo nel caso in cui le istruzioni SIMD sono realmente utilizzabili, ovvero solo nei casi in cui possiamo veramente gestire più blocchi di dati all'unisono.

Le funzioni che presentavano sufficienti caratteristiche per poter poi essere ottimizzate tramite istruzioni SIMD sono state le seguenti:

- *riempi_out_con_meno_due*;
- *distanza_euclidea*.

In realtà si è implementata anche la funzione *distanza_euclidea_rispetto_a* tramite istruzioni SIMD, ma si è osservato che queste ultime non apportavano alcuna miglioria; dunque, si è deciso di escludere questa feature dall'artefatto finale. In particolare, quest'ultima implementazione non apportava alcuna miglioria perché uno dei due punti su cui si calcolava la distanza non era (e non poteva essere) allineato in memoria, dunque non era possibile prelevare blocchi di dati continui senza causare una Segmentation Fault.

Per quanto riguarda *riempi_out_con_meno_due*, nell'implementazione a 64bit risulta molto più efficiente, perché in essa è possibile lavorare su ben otto interi per volta, mentre nell'implementazione a 32bit è possibile lavorare su quattro interi per volta. Per quanto riguarda *distanza_euclidea*, purtroppo, non ci sono considerazioni di questo tipo da sollevare. Nell'implementazione del codice assembly a 32bit si è fatto uso dei registri dello stack a 8 livelli del coprocessore matematico x87 della CPU, così da poter restituire in modo pulito ed intuitivo i risultati in virgola mobile, senza dover ricorrere al passaggio di parametri per riferimento. Invece, nell'implementazione a 64bit, il registro XMM0 è stato utilizzato per restituire i valori in virgola mobile a doppia precisione.

Osserviamo come non si è fatto uso dei registri ZMM dell'estensione AVX512 nell'implementazione a 64bit, in quanto non era noto se una tale pratica potesse essere ritenuta scorretta.

In conclusione, si nota come non si è fatto ricorso, nella versione a 64bit, delle istruzioni *pushaq* e *popaq*, in quanto quest'ultima, contrariamente a quanto descritto nella documentazione del processore, ripristina il valore dei registri XMM a zero, impedendo una restituzione pulita dei risultati calcolati.

VII. Confronti tra le varie implementazioni

Tutti i seguenti test sono stati effettuati tramite le soluzioni a 32bit.

- Dataset 5000x50:
  - soluzione senza istruzioni SIMD e senza Ball Tree (ricerca esaustiva):

    ```
    N:5000 , d:50 , k:5000
    CFS time = 7.508 secs
    ```

  - soluzione senza istruzioni SIMD e con Ball Tree:

    ```
    N:5000 , d:50 , k:5000
    CFS time = 5.472 secs
    ```

  - soluzione con istruzioni SIMD e con Ball Tree:

    ```
    N:5000 , d:52 , k:5000
    CFS time = 1.138 secs
    ```

    (si nota come in questa soluzione d=52 e non d=50, in realtà *d* è pari a 50, ma come spiegheremo nell'appendice dedicata ad i template, viene considerata pari a 52 per evitare problemi di segmentation fault; stesse considerazioni valgono per gli altri test)
  - soluzione sklearn:

    ```
    C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
    La funzione ha impiegato 0.7522118091583252 secondi per essere eseguita.
    ```

- Dataset 25000x20:
  - soluzione senza istruzioni SIMD e senza Ball Tree (ricerca esaustiva):

    ```
    N:25000 , d:20 , k:25000
    CFS time = 91.539 secs
    ```

  - soluzione senza istruzioni SIMD e con Ball Tree:

    ```
    N:25000 , d:20 , k:25000
    CFS time = 73.555 secs
    ```

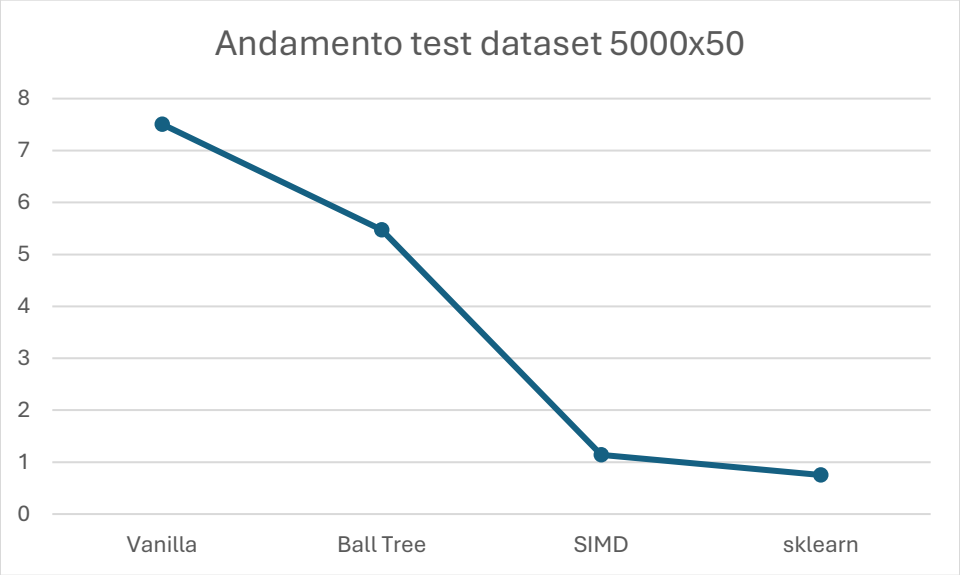
  - soluzione con istruzioni SIMD e con Ball Tree:

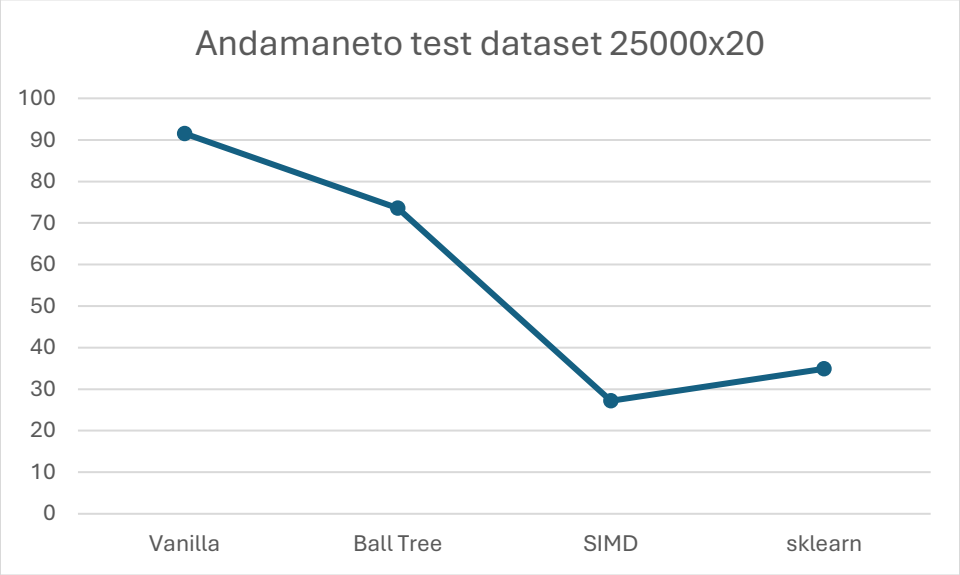
    ```
    N:25000 , d:20 , k:25000
    CFS time = 27.203 secs
    ```

- soluzione sklearn:

```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
La funzione ha impiegato 34.88030934333801 secondi per essere eseguita.
```

Si può osservare come nell'ultimo test, si è riusciti a "battere" le prestazioni di sklearn. Concludiamo mostrando l'andamento grafico dei test:





Possiamo notare, dai grafici, come la derivata prima destra del secondo punto da sinistra sia, in valore assoluto, molto più grande. Ciò indica che le ottimizzazioni tramite istruzioni SIMD hanno avuto un impatto maggiore rispetto alle ottimizzazioni ottenute tramite l'uso di un Ball Tree come struttura dati.

In allegato sono inclusi gli script python (rispettivamente a 32 e 64 bit) utilizzati per generare i dataset ed i confronti con sklearn.

VIII. Ottimizzazioni mancate

Un'ulteriore ottimizzazione, che però non è stato possibile implementare, consiste nel definire una matrice quadrata di ordine pari al numero di punti presenti nel dataset, contenete nella cella ij-esima la distanza euclidea tra i punti *i* e *j*. Questa ottimizzazione avrebbe "ammazzato" i tempi, proprio perché sarebbe andata ad azzerare (o quasi) il tempo unitario di riferimento del nostro studio di complessità; infatti, durante la trattazione, ogni volta che si faceva riferimento ad un costo computazionale, abbiamo dato per scontato che la distanza euclidea fosse l'operazione di costo unitario. È immediato comprendere che una tale ottimizzazione avrebbe portato uno speedup della soluzione enorme.

Come accennato ad inizio paragrafo, non è stato possibile apportare tale ottimizzazione perché il costo spaziale dovuto alla memorizzazione della matrice descritta era troppo grande. Per dare un'idea, per un dataset di soli cinquantamila punti tale matrice avrebbe richiesto ben *10GB* di spazio (*2.500.000.000* celle da *32/64 bit* l'una). Dunque, si lascia al lettore l'idea di tale ottimizzazione sottolineando come la fattibilità di quest'ultima dipendi unicamente dalle risorse computazionali del calcolatore, per il resto l'implementazione è banale.

IX. OpenMp

L'utilizzo di OpenMP (Open Multi-Processing) permette di parallelizzare sezioni di codice tramite l'utilizzo di direttive. Le direttive utilizzate (descritte all'interno dei codici delle soluzioni) ci hanno permesso di parallelizzare la funzione *changeClId_for_seeds* (ovvero la funzione che assegna le etichette ai vari nodi) e di parallelizzare la query (ricorsiva) sul BallTree, proteggendo dalle race condition l'accesso concorrente alla variabile di tipo *Punti* condivisa dai vari thread.

Si può notare come in questa relazione manchino i dati che mostrano l'incremento di prestazioni ottenuto grazie all'utilizzo di OpenMp, e non a caso. Infatti, utilizzando il meccanismo fornito nei template, non è possibile misurare il tempo reale di esecuzione. Sarebbe possibile utilizzando le funzioni offerte dalla libreria di OpenMp stessa, ma si è

preferito non utilizzarle, sia per non modificare di troppo il template fornito, sia perché anche se le avessimo utilizzate non avremmo potuto comunque essere in grado di osservare l'incremento di prestazioni reale, ciò perché l'ambiente di testing utilizzato è quello fornito da una semplice macchina virtuale, che può tranquillamente simulare l'utilizzo di più thread su un singolo thread fisico. Dunque, tutte le considerazioni che avremmo potuto fare nel caso dell'utilizzo delle funzioni offerte dalla libreria OpenMp (per cronometrare il tempo impiegato) sarebbero comunque state molto lasche, mentre la corrente trattazione cerca di offrire spunti e osservazioni molto pesate.

Appendice (a)

In quest'appendice discuteremo brevemente delle modifiche apportate al template fornito.

La prima modifica consiste nell'inserire nella struct *params* l'attributo *MinPts* e *Eps*, in quanto, per definizione, fanno parte dell'input dell'algoritmo. Dopodiché, si ha modificato il sistema di lettura degli input, inserendo controlli dedicati a questi ultimi due parametri più la loro formattazione in, rispettivamente, valori interi e float/double a partire da caratteri ascii. Ovviamente, nella funzione *main* è stata aggiunta la chiamata alla funzione riguardante l'implementazione dell'algoritmo, ovvero alla funzione *DBSCAN*. Inoltre, sempre nella funzione *main*, è stata inserita una riga di codice che sovrascrive il valore del parametro di input *k* con il valore del parametro di input *N*, nonostante sia sempre necessario che l'utente inserisca un valore per il parametro *k* in input. Inoltre, sempre nella funzione *main*, è stata mitigata la procedura di lettura del file con estensione *.labels* con le rispettive procedure di controllo di corretta formattazione. Quest'ultima modifica è stata resa necessaria per semplificare una modifica molto più importante, ovvero la modifica della funzione *load_data*. All'interno di quest'ultima funzione è stata inserita la procedura *adjust_matrix* che ha il compito di portare il parametro di input *d* ad essere sempre un multiplo di *4*. Più nello specifico, se il numero di colonne della matrice *ds* risulta non essere un multiplo di *4*, allora si aggiunge il numero minimo di colonne tali per cui questa condizione non vale più (ovviamente le colonne aggiunte conterranno valori nulli, così da non alterare il calcolo della distanza euclidea). Ciò è stato reso necessario perché la matrice *ds* viene memorizzata come un vettore secondo un formato row-major order, però tale matrice viene anche memorizzata in modo "allineato" (così da poter applicare le istruzioni SIMD) ma essendo il vettore la vera entità allineata (e non la matrice) si

ha che le singole righe della matrice non è detto che siano allineate; dunque, si assicura che le colonne siano in numero multiple di *4* così da assicurare l'allineamento di ogni singola riga.

Bibbliografia

[1]. Ester, Martin, et al. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise." KDD '96 Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, AAAI Press, 1996, pp. 226-231.