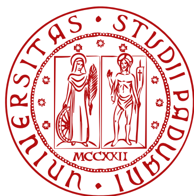


Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Laurea in Ingegneria dell'Informazione

Algoritmi distance-based per l'identificazione di outlier

Relatore
Prof. Andrea Alberto Pietracaprina

Studente
Bartolomeo Morellato

29 Settembre 2023

Anno accademico 2022/2023

"Nessun insegnamento vale quanto l'esempio"

Robert Baden-Powell
Il libro dei Capi

A tutte le persone che mi sono state vicine in questo percorso,
al capannone di Pietro, la migliore aula studio di Padova

Abstract

Questa tesi di laurea studia i principali algoritmi di rilevamento di distance-based outlier noti in letteratura. L'obiettivo principale è comprendere le definizioni di outlier adottate, studiare le strategie algoritmiche utilizzate e fornire una dettagliata analisi della complessità degli algoritmi. In una prima parte si fornisce una definizione della nozione di distance-based outlier e il problema del rilevamento di outlier. Successivamente, vengono descritti tre algoritmi proposti in due articoli, esplicitando le scelte implementative relative ad alcuni passaggi chiave e alle strutture dati da utilizzare. Per ciascun algoritmo viene effettuata l'analisi della complessità in termini di numero di operazioni e, in alcuni casi, di numero di blocchi trasferiti da disco a RAM e viceversa. Per un algoritmo viene anche fornita una versione che ne migliora notevolmente l'efficienza a scapito di una moderata perdita di accuratezza negli outlier identificati.

Indice

Indice **vii**

1 Introduzione **1**

2 Nozioni Preliminari **3**
- 2.1 Modello di calcolo 3
- 2.2 Dataset e Outlier 4
- 2.3 Funzioni Ausiliarie 5
- 2.4 Algoritmo Semplice 5

3 Algoritmo Nested Loop **9**
- 3.1 Definizioni 9
- 3.2 Algoritmo 11
- 3.3 Analisi di complessità 12
- 3.4 Considerazioni 14

4 Algoritmo FindAllOutsM **15**
- 4.1 Definizioni 15
- 4.2 Algoritmo 19
- 4.3 Analisi di complessità 21
- 4.4 Generalizzazione di FindAllOutsM a dimensioni > 2 22
- 4.5 Considerazioni 23

5 Algoritmo Solving Set **25**
- 5.1 Definizioni 25
- 5.2 Algoritmo 28
- 5.3 Analisi di complessità 30
- 5.4 Considerazioni 31

Bibliografia **33**

Capitolo 1

Introduzione

Un *outlier* è un oggetto in un dataset che si differenzia molto rispetto agli altri oggetti del dataset.

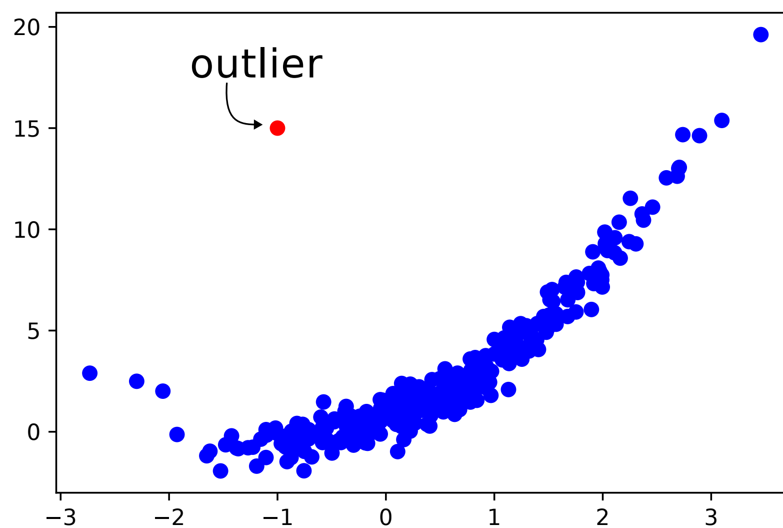


Figura 1.1: Esempio di outlier in un dataset bidimensionale.

L'identificazione efficiente di outlier in grandi dataset è un argomento di ricerca attivo nell'ambito del data mining che ha molti risvolti pratici. Infatti l'identificazione di outlier, oltre ad essere essenziale nella fase di "pulitura" dei dati nel machine learning, trova molte applicazioni negli ambiti in cui è utile riconoscere comportamenti anormali e/o illegali, come ad esempio l'identificazione di intrusioni su una rete di computer, identificazioni di frodi, furti d'identità, eccetera. Inoltre, la ricerca di outlier può fornire l'intuizione per lo sviluppo di nuove teorie in settori che riguardano, ad esempio, le diagnosi mediche, ricerche di marketing, analisi delle prestazioni

di atleti. È in questi ambiti che gli eventi rari sono più interessanti degli eventi comuni ed è per questo che l'identificazione di outlier è un'attività che sta diventando sempre più importante.

Questa tesi ha come obiettivo quello di esporre in modo chiaro e rigoroso gli algoritmi presentati in [1] e [2]. Partendo dalle diverse definizioni di outlier adottate nei due articoli in studio, verranno esposti gli algoritmi in forma di pseudocodice e verrà studiata la loro complessità.

L'organizzazione della tesi è la seguente.
Nel Capitolo 2 vengono presentati il modello di calcolo adottato in questo elaborato, il concetto di outlier e alcune nozioni base che riguardano la struttura del dataset e degli oggetti che ne fanno parte. Viene anche riportato l'algoritmo *Algoritmo Semplice* (Sezione 2.4): un algoritmo immediato che risolve in modo banale il problema dell'identificazione di outlier.
Il Capitolo 3 tratta l'algoritmo *Nested Loop*, ovvero un algoritmo che ha work uguale all'algoritmo Semplice, ma che per dataset molto grandi ha un risparmio considerevole sulla complessità I/O.
Nel Capitolo 4 viene introdotto l'algoritmo *FindAllOutsM*, il quale, sfruttando una struttura a celle (ossia dei quadrati che partizionano il dataset) riesce ad avere un work migliore rispetto agli altri algoritmi.
Infine, il Capitolo 5 analizza l'algoritmo *Solving Set* (presentato in [2]). Questo non solo si occupa dell'identificazione degli outlier, ma si estende anche alla previsione di nuovi outlier, affrontando entrambi i problemi in modo efficiente.

Capitolo 2

Nozioni Preliminari

In questo capitolo vengono presentate delle definizioni e risultati preliminari utili a chiarificare notazione e strumenti di base usati poi nella trattazione.

2.1 Modello di calcolo

Un modello di calcolo (o modello di computazione) è utilizzato per lo sviluppo e l'analisi di algoritmi ed è costituito da tre parti: il modello architetturale, il modello di programmazione ed il modello di costo.

In questa tesi si fa riferimento sia al modello *RAM* (*Random Access Machine*) tradizionale che al modello di calcolo *Disk Model*. A differenza della più semplice architettura del modello *RAM*, dove è presente solo una CPU collegata direttamente ad una memoria RAM (Random Access Memory); l'architettura del Disk Model (DM da ora in poi) è composta da una CPU che ha accesso diretto a una memoria RAM, di dimensione G parole, dove è memorizzato l'algoritmo e dove vengono svolte le operazioni. È presente inoltre un disco di capacità virtualmente illimitata, collegato direttamente alla RAM, dove vengono memorizzati i dati che non possono essere salvati nella RAM. I trasferimenti di dati tra RAM e disco avvengono in blocchi di B parole.

Un algoritmo nel DM è un una sequenza di istruzioni appartenenti allo stesso insieme di istruzioni del modello RAM (ad esempio confronti, operazioni aritmetico/logiche, letture/scritture da/su la memoria RAM) al quale si aggiungono due istruzioni che permettono il trasferimento di un blocco da disco a RAM (Input) e viceversa (Output). Si distinguono quindi due tipi di operazioni: operazioni elementari e operazioni di Input/Output.

Ne segue che le prestazioni degli algoritmi nel DM sono valutate principalmente attraverso due metriche temporali: il "Work" (numero di operazioni elementari) e la "Complessità di I/O" (numero di operazioni di Input/Output). Entrambe queste metriche dipendono dalla dimensione dell'istanza (N), dalla dimensione della RAM (G) e dalla dimensione dei blocchi di trasferimento (B).

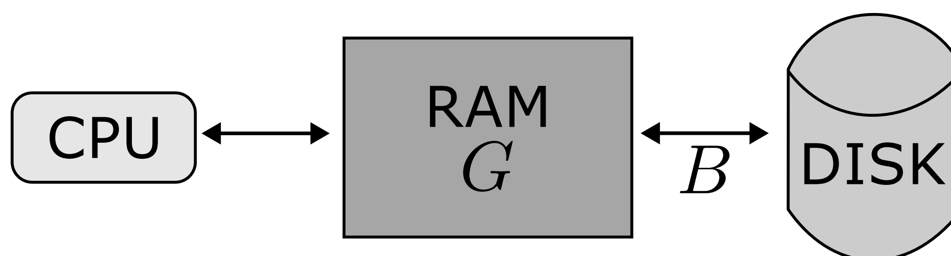


Figura 2.1: Architettura Disk Model

2.2 Dataset e Outlier

In questa tesi un dataset è una collezione di oggetti, detti dati, dove ogni oggetto è rappresentato da un vettore di dimensione d. Dato un dataset T, l'oggetto $a \in T$ è il vettore $\vec{a} = (\alpha_1, \alpha_2, ..., \alpha_d) \in \mathbb{R}^d$. Ne segue che la dimensione di un dataset è la dimensione d dei suoi oggetti.

Precedentemente è stata introdotto informalmente il concetto di outlier come "un oggetto del dataset che si distanzia di molto dagli altri". Per poter lavorare con questi oggetti, dobbiamo introdurre una definizione più rigorosa; usiamo quindi quella di DB-outlier (Distance-Based outlier) presentata in [1].

Definizione 2.1 (DB(s, r)-outlier)**.** Sia dato un dataset T, una distanza $r \in \mathbb{R}_{\geq 0}$ e $s \in (0, 1)$. Un oggetto $p \in T$ è un DB(s, r)-outlier se una frazione $> s$ di oggetti di T si trova ad una distanza $> r$ da p.

Detto in altre parole, un oggetto p in T è un DB(s, r)-outlier se all'esterno della circonferenza di raggio r centrata in p, ci sono almeno $s \cdot N$ oggetti, dove N è il numero di oggetti del dataset. In particolare, si noti che per essere un DB(s, r)-outlier un oggetto di T può avere al massimo $M = N(1 - s)$ oggetti entro raggio r.

Per il problema dell'identificazione di outlier è necessario definire il concetto di distanza tra due oggetti.

Definizione 2.2 (Distanza)**.** Dato un dataset T, la distanza tra $a, b \in T$ è la distanza Euclidea tra i vettori $\vec{a}$ e $\vec{b}$; ovvero

$$\sqrt{\sum_{i=1}^{d} (\alpha_i - \beta_i)^2}$$

Nota 2.3 (Restrizioni a spazi Euclidei)**.** Per concretezza, in questa tesi ci si è ristretti a dataset facenti parte di uno spazio Euclideo. Gli algoritmi che saranno presentati in seguito, sono però applicabili anche a dataset che appartengono a spazi metrici più generali.

A un livello di astrazione più basso, un oggetto del dataset è caratterizzato da d campi che rappresentano le coordinate dello spazio Euclideo e da un campo booleano chiamato *isOutlier* che vale *true* se l'oggetto è un outlier e *false* se l'oggetto non è un outlier (di default tutti gli oggetti hanno questo campo settato a *NULL*). Si indica con *p.isOutlier* il campo *isOutlier* dell'oggetto p.
In questa tesi si considera il dataset come un vettore che memorizza tutti gli oggetti che appartengono ad esso. Per semplicità da ora in poi l'i-esimo oggetto del dataset T è indicato con p_i.

Nota 2.4 (Allocamento del dataset nel DM)**.** Supponendo di lavorare con dataset molto grandi, si può affermare che $N \gg G$, dove N è il numero di oggetti del dataset e G è il numero di oggetti che possono essere memorizzati nella RAM. Facendo riferimento al modello DM, è ragionevole pensare che l'intero dataset sia memorizzato su disco e, solo durante l'esecuzione dell'algoritmo, venga caricato sulla RAM a blocchi di B parole.

2.3 Funzioni Ausiliarie

Negli algoritmi presentati vengono utilizzate numerose funzioni ausiliarie. Vengono qui riportate tre funzioni che compaiono in tutti gli algoritmi.

- La funzione `distance`(p, q) calcola la distanza Euclidea tra i due oggetti $p, q \in T$.
  Come detto in precedenza, la distanza tra due oggetti $\in T$ è la distanza Euclidea tra gli oggetti (definizione 2.2 a pagina 4). Per un dataset di dimensione d, si assume che il costo del calcolo di una distanza tra due oggetti sia $\Theta(d)$.

- La funzione `markNotOutlier`(p) segna l'oggetto $p \in T$ come NON-DB(s, r)-outlier mettendo il campo *p.isOutlier* uguale a *false*.
  Questa operazione ha costo costante $O(1)$.

- La funzione `markOutlier`(p) segna l'oggetto $p \in T$ come DB(s, r)-outlier mettendo il campo *p.isOutlier* uguale a *true*.
  Questa operazione ha costo costante $O(1)$.

2.4 Algoritmo Semplice

Viene ora presentato un algoritmo banale in grado di risolvere il problema dell'identificazione di outlier. Come verrà mostrato in seguito, questo algoritmo ha un work quadratico rispetto al numero di elementi del dataset e quindi per dataset molto grandi l'algoritmo non è utilizzabile nella pratica.

Algoritmo 1: Algoritmo Semplice(T, N, s, r)

Input: il dataset T, il numero N di oggetti in T, la frazione s, e la distanza r.

Output: ogni oggetto in T è "segnato" o come DB(s, r)-outlier oppure come NON-DB(s, r)-outlier.

```
1  M ← N(1 − s);
2  for (i = 0; i < N; i++) do
3      count ← 0;
4      for (j = 0; 1 < N; j++) do
5          if (distance(p_i, p_j) ≤ r) then
6              count + +;
7              if (count > M) then
8                  markNotOutlier(p_i);
9                  break;
10             end
11         end
12         markOutlier(p_i);
13     end
14 end
```

L'algoritmo fa uso di tre funzioni ausiliarie (già analizzate a pagina 5): `distance`(p, q), `markNotOutlier`(p) e `markOutlier`(p).

Work

Il ciclo **for** esterno viene eseguito esattamente N volte. Ad ogni iterazione di questo ciclo viene inizializzata la variabile $count_p$ e viene svolto il ciclo interno. Il ciclo **for** interno viene eseguito al più N volte per ogni iterazione del ciclo esterno. Ad ogni iterazione di questo ciclo viene calcolata `distance`(p_i, p_j) (costo $\Theta(d)$), tutte le altre operazioni hanno costo costante.
Ne segue che la complessità al caso peggiore dell'algoritmo è $O(dN^2)$.

Complessità I/O

L'algoritmo trova gli outlier semplicemente confrontando ogni oggetto del dataset con ogni altro oggetto del dataset. Per eseguire il calcolo della distanza oggetto-oggetto, l'intero dataset (che risiede sul disco) deve essere caricato sulla RAM. Il trasferimento di dati da disco a RAM è eseguito a blocchi di B parole, quindi si può considerare il dataset diviso in $n = \frac{N}{B}$ blocchi.
Vale che $\forall\, p \in T$ l'algoritmo carica in RAM n blocchi. Ne segue che la complessità I/O è $\Theta(N\frac{N}{B}) = \Theta(\frac{N^2}{B})$.

Considerazioni

L'elevata complessità I/O e l'elevato work rendono questo algoritmo inutilizzabile nella pratica. Nei capitoli successivi si vedrà come alcune intuizioni possono portare ad algoritmi capaci di identificare gli outlier che hanno complessità migliori dell'algoritmo semplice.
In [1], l'algoritmo viene denominato *Index Based* poiché si avvale di strutture di indicizzazione, come R-tree e kd-tree, per ottimizzare la ricerca dei vicini di un oggetto. È degno di nota riportare che, utilizzando queste strutture, è possibile migliorare l'efficienza dell'algoritmo. È importante sottolineare, però, che per dataset ad alta dimensionalità, la complessità rimane comunque quasi quadratica.

Capitolo 3

Algoritmo Nested Loop

In questo capitolo viene studiato l'algoritmo Nested Loop e l'analisi della sua complessità, seguendo la loro presentazione fatta in [1]. Questo è un algoritmo orientato a blocchi e basato su cicli innestati che ha come obiettivo quello di individuare gli outlier di un dataset. Particolare importanza verrà data all'analisi della complessità di I/O, in quanto si ha un netto miglioramento rispetto all'algoritmo semplice.

L'algoritmo fa uso di due array, $Arr1$ e $Arr2$, localizzati nella RAM. Durante le varie iterazioni del ciclo, l'intero dataset viene caricato nei due array e viene calcolata la distanza per ogni coppia di oggetti nei due array. Per ogni oggetto p di $Arr1$ viene memorizzato il numero di *r-neighbours* (vicini in una circonferenza di raggio r), e se questo numero supera $M = N(1-s)$ il conteggio viene fermato e p viene dichiarato NON-DB(s, r)-outlier. Se invece, dopo aver confrontato l'oggetto p con tutti gli altri oggetti del dataset, il numero di *r-neighbours* di p è minore di M, p viene dichiarato DB(s, r)-outlier.

3.1 Definizioni

Viene usata la definizione di DB-Outlier presentata in [1], già discussa nel capitolo precedente.

Con lo scopo di rendere il più chiaro possibile lo studio di questo algoritmo, è necessario capire meglio come e quando gli oggetti del dataset vengono trasferiti dal disco alla RAM. Come illustrato in figura 2.1, la memoria RAM può ospitare fino a G oggetti del dataset, e i trasferimenti da disco a RAM avvengono in blocchi di B oggetti. L'algoritmo usa due array, chiamati $Arr1$ e $Arr2$, nei quali (nel corso delle varie iterazioni) viene caricato l'intero dataset. In quanto i due array sono localizzati nella RAM, ognuno di essi potrà contenere al massimo $\frac{G}{2}$ oggetti. Supponendo che $B = \frac{G}{2}$ segue che per popolare un array è necessario quindi un solo trasferimento di un blocco (ovvero una sola operazione di I/O).

In sintesi, si può considerare il dataset T suddiviso in $n = \frac{N}{B}$ blocchi, dove l'i-esimo blocco del dataset contiene gli oggetti p_j con $i \cdot B \leq j < (i+1) \cdot B$, ed è indicato con B_i.

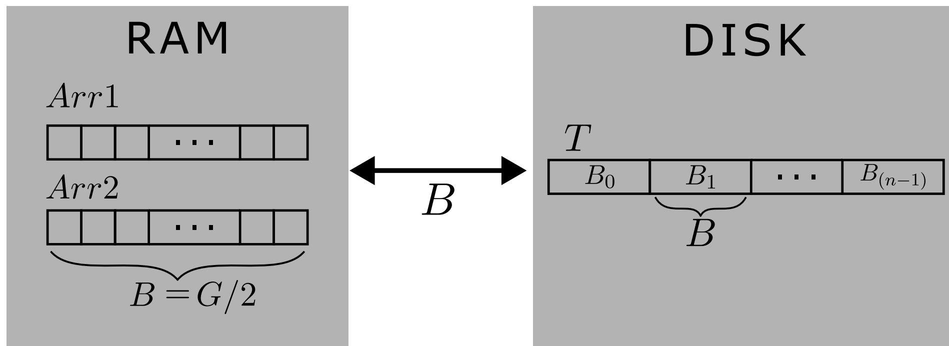


Figura 3.1: Architettura DM in Nested Loop

3.2 Algoritmo

Algoritmo 2: Nested Loop(T, N, s, r, B)

Input: il dataset T, il numero N di oggetti in T, la frazione s, la distanza r e il numero di oggetti in un blocco B.

Output: ogni oggetto in T è "segnato" o come DB(s, r)-outlier oppure come NON-DB(s, r)-outlier.

```
1  M ← N(1 − s);
2  n ← N/B;
3  Arr1 ← B_{n−1};
4  Arr2 ← Arr1;
5  for (i = n − 1; i ≥ 0; i--) do
6      swap(Arr1, Arr2);
7      for each (p_l in Arr1)
8          count[l] ← 0;
9          for each (p_m in Arr1)
10             if (distance(p_l, p_m) ≤ r) then
11                 count[l]++;
12                 if (count[l] > M) then
13                     markNotOutlier(p_l);
14                     procedi con il prossimo p_l;
15                 end
16             end
17         end
18     end
19     for (j = ((i + 1) mod n); (j mod n) ≠ i; j++) do
20         Arr2 ← B_{(j mod n)};
21         for each (p_l non segnato in Arr1)
22             for each (p_m in Arr2)
23                 if (distance(p_l, p_m) ≤ r) then
24                     count[l]++;
25                     if (count[l] > M) then
26                         markNotOutlier(p_l);
27                         procedi con il prossimo p_l;
28                     end
29                 end
30             end
31         end
32     end
33     for each (p_l non segnato in Arr1)
34         markOutlier(p_l);
35     end
36 end
```

L'algoritmo fa uso di quattro funzioni ausiliarie: la funzione `distance`(p, q), `markNotOutlier`(p), `markOutlier`(p), ed infine la funzione `swap`$(Arr1, Arr2)$ che scambia i nomi dei 2 array, e che quindi ha costo costante.

3.3 Analisi di complessità

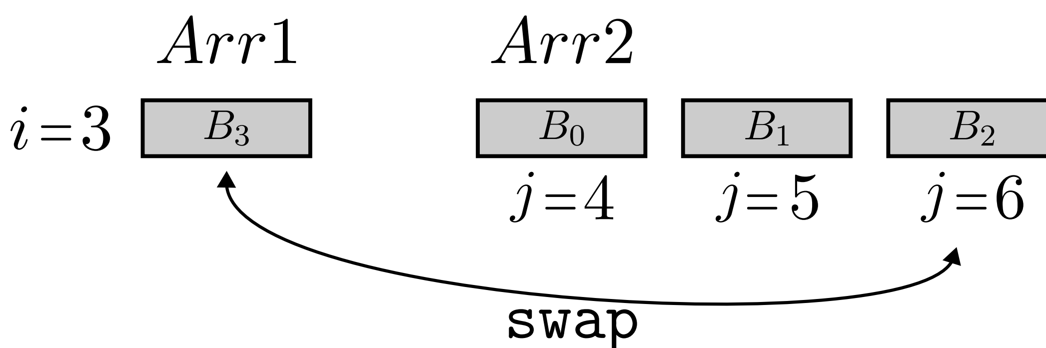
Work

Ricordando che la funzione `distance`(p, q) ha complessità $\Theta(d)$, dove d è la dimensione del dataset, è facile vedere che la complessità dell'algoritmo è $O(dN^2)$, in quanto per ogni oggetto in T vengono calcolate, al caso peggiore, tutte le distanze con gli altri $N - 1$ oggetti (è questo il caso in cui p è un DB(s, r)-outlier).

Complessità I/O

Se, per quanto riguarda il work l'algoritmo Nested-Loop e l'algoritmo Semplice sono equivalenti, per quanto concerne la complessità I/O l'algoritmo Nested-Loop è migliore di quello Semplice di un fattore B.

Per rendere più comprensibile come l'algoritmo carica il dataset nella RAM viene fornito il seguente esempio:
Sia $B = N/4$, ne segue che il dataset è diviso in 4 blocchi A, B, C, D, dove ognuno contiene $\frac{1}{4}$ del dataset.
Con l'aiuto delle figure sottostanti, dove i blocchi che vengono caricati nella RAM sono colorati in grigio, si mostra il funzionamento dello pseudocodice:



Nella prima iterazione del ciclo esterno, viene caricato il blocco B_3 in $Arr1$ e nelle tre iterazioni del ciclo interno vengono caricati i blocchi B_0, B_1, B_2; vengono quindi eseguite 4 operazioni di I/O. Dopodiché si scambiano i nomi dei due array e si avrà salvato in $Arr1$ il blocco B_2 e in $Arr2$ il blocco B_3, senza compiere alcuna operazione di I/O.

Nella seconda iterazione del ciclo esterno si ha già in $Arr1$ il blocco B_2 e in $Arr2$ il blocco B_3. Quindi vengono compiute solo due operazioni di I/O per caricare in $Arr2$ i blocchi B_0 e B_1. Dopo lo swap si ha in $Arr1$ il blocco B_1 e in $Arr2$ il blocco B_2, senza compiere alcuna operazione di I/O.

Analogamente a quanto detto sopra, nella terza iterazione del ciclo esterno vengono eseguite due operazioni di I/O.

Anche nell'iterazione finale del ciclo esterno (ovvero per $i = 0$) vengono svolte due operazioni di I/O per caricare i blocchi B_2 e B_3 in $Arr2$.

In totale si ha eseguito 10 caricamenti di blocchi (quindi 10 operazioni di I/O), ovvero si ha effettuato $\frac{10}{4} = 2.5$ caricamenti dell'intero dataset.

Nel caso più generale, vale la seguente proprietà:

Proposizione 3.1. *Se un dataset T è diviso in $n = \lceil \frac{N}{B} \rceil$ blocchi, allora*

(i) *negli array vengono caricati al più $n + (n-2)(n-1)$ blocchi;*

(ii) *il dataset viene caricato interamente $\geq n-2$ volte.*

Dimostrazione. (i) Ognuno degli n blocchi viene caricato esattamente una volta nella prima iterazione del ciclo. Alla fine di ogni iterazione rimangono salvati nella RAM due blocchi, quindi è necessario caricare solo altri $n-2$ blocchi dalla seconda iterazione in poi. Quindi negli array vengono caricati $n + (n-2)(n-1)$ blocchi.
(ii) Il numero di caricamenti totale del dataset è quindi:
$\frac{n+(n-2)(n-1)}{n} = \frac{n^2-2n+2}{n} = n - 2 + \frac{2}{n} \geq n-2.$ □

La complessità I/O di Nested Loop è quindi $O(n+(n-2)(n-1)) = O(n^2-2n+2) = O(n^2)$. Ricordando che $n = \frac{N}{B}$, la complessità si può esprimere anche come $O\left(\left(\frac{N}{B}\right)^2\right)$.

3.4 Considerazioni

Confrontando l'algoritmo Nested Loop con l'Algoritmo Semplice (studiato alla Sezione 2.4), si vede come i due algoritmi abbiano work quadratico rispetto a N. Si potrebbe essere tentati di dire che i due algoritmi siano intercambiabili. Nella pratica, però, il trasferimento di un blocco da disco a RAM è ordini di grandezza più costoso di un'operazione eseguita su dati in RAM. La complessità I/O acquista quindi una notevole importanza nell'analisi di algoritmi che devono elaborare input di grandi dimensioni. Ricordando che la complessità I/O dell'Algoritmo Semplice è $O\left(\frac{N^2}{B}\right)$ e quella di Nested Loop è $O\left(\left(\frac{N}{B}\right)^2\right)$ si vede come l'algoritmo Nested Loop sia migliore di un fattore B, rendendo l'algoritmo più efficiente.

Capitolo 4

Algoritmo FindAllOutsM

In questo capitolo viene presentato un algoritmo che, anziché guardare ai singoli oggetti del dataset, partiziona lo spazio in celle (che rispettano certe proprietà), mappa gli oggetti del dataset nelle celle giuste, e, facendo considerazioni su queste ultime, riesce ad identificare i DB(s, r)-outlier del dataset.
Nella prima parte del capitolo si definisce il concetto di cella e le sue proprietà. Successivamente viene analizzato l'algoritmo FindAllOutsM, presentato in [1], mettendo in luce un errore fatto nell'analisi della sua complessità, e quindi fornendo un'analisi più rigorosa. Viene poi notato che rilassando la definizione di outlier, si può risolvere il problema dell'identificazione di outlier in un tempo ancora migliore di FindAllOutsM.

4.1 Definizioni

Viene usata la definizione di DB Outlier presentata in [1], già discussa nei capitoli precedenti.

Per eliminare la complessità quadratica rispetto al numero di oggetti del dataset, l'algoritmo FindAllOutsM fa uso delle celle.
Si presentano ora la nozione di cella e le sue proprietà, limitandosi al caso di un dataset bidimensionale.
Senza perdere di generalità si può supporre che tutti gli oggetti del dataset si trovano nel primo quadrante, ossia il quadrante definito dagli oggetti che hanno coordinate ≥ 0.

Definizione 4.1 (Cella)**.** Dato un spazio bidimensionale, una cella è un quadrato di lato $l = \frac{r}{2\sqrt{2}}$. L'insieme delle celle partiziona l'intero spazio.
Si indichi con $C_{x,y}$ la cella che sta sulla riga x e sulla colonna y. In particolare la cella $C_{0,0}$ è quella il cui angolo in basso a sinistra si trova sul punto $(0, 0)$.

L'insieme delle celle è quindi un rettangolo che giace sul primo quadrante e che include tutto il dataset.

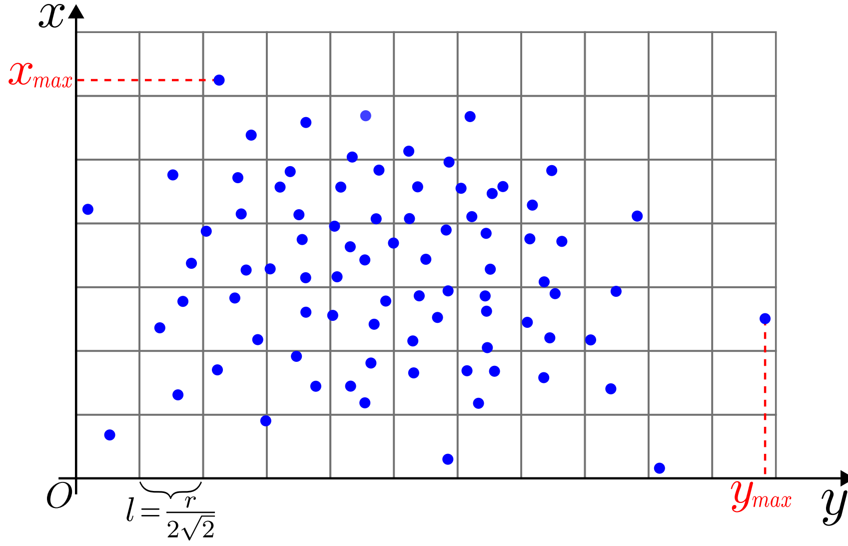


Figura 4.1: Dataset e celle giacenti sul primo quadrante

Nota 4.2. La cella $C_{x,y}$ contiene al suo interno tutti gli oggetti che hanno coordinate (u, v), dove $xl \leq u < (x + 1)l$ e $yl \leq v < (y + 1)l$.

Nota 4.3. Per semplicità si suppone di conoscere x_{max}, ovvero il massimo valore delle ascisse, e y_{max}, ovvero il massimo valore delle ordinate, di un oggetto del dataset. È facile ora ottenere il numero m di celle che partizionano il dataset:

$$m = \left\lceil \frac{x_{max}}{l} \right\rceil \cdot \left\lceil \frac{y_{max}}{l} \right\rceil$$

In molti casi pratici, scelte ragionevoli assicurano che $m \ll N$, ma è doveroso dire che nel caso in cui gli oggetti del dataset siano pochi e molto sparsi si avrà $m > N$ e la maggior parte delle celle sarebbe vuota. Si noti anche come $m = R \cdot C$, dove R è il numero di righe in cui è suddiviso l'insieme delle celle e C è il numero delle colonne.

Definizione 4.4 (L_1-*neighbours*)**.** Data una cella $C_{x,y}$ l'insieme L_1-*neighbours* (dove L_1 è l'abbreviazione di *Layer 1*) di $C_{x,y}$ è l'insieme delle celle immediatamente vicine a $C_{x,y}$; ovvero:

$$L_1(C_{x,y}) = \{C_{u,v} \,|\, u = x \pm 1, v = y \pm 1, C_{u,v} \neq C_{x,y}\}$$

Per una cella vale che $|L_1(C_{x,y})| \leq 8$.

Proprietà 4.5. La distanza tra ogni coppia di oggetti nella stessa cella è al massimo $\frac{r}{2}$.

Dimostrazione. In quanto diagonale di una cella misura $\sqrt{2}l = \sqrt{2}\frac{r}{2\sqrt{2}} = \frac{r}{2}$ e due oggetti nella stessa cella sono alla massima distanza se sono posizionati in due angoli opposti della cella. □

Proprietà 4.6. Se $C_{u,v}$ è una L_1-*neighbour* di $C_{x,y}$, allora la distanza tra ogni oggetto $p \in C_{u,v}$ e ogni oggetto $q \in C_{x,y}$ è al massimo r.

Dimostrazione. In quanto la distanza tra una qualsiasi coppia di oggetti in due celle diverse non può essere maggiore di due volte la lunghezza della diagonale. □

Definizione 4.7 (L_2-*neighbours*)**.** Data una cella $C_{x,y}$ l'insieme $L_2 - neighbours$ (dove L_2 è l'abbreviazione di *Layer 2*) di $C_{x,y}$ è l'insieme:

$$L_2(C_{x,y}) = \{C_{u,v} \,|\, x-3 \leq u \leq x+3, y-3 \leq v \leq y+3, C_{u,v} \notin L_1(C_{x,y}), C_{u,v} \neq C_{x,y}\}$$

Per una cella vale che $|L_2(C_{x,y})| \leq 7^2 - 3^2 = 40$.

Nota 4.8. È facile vedere che se $C_{i,j} \in L_2(C_{x,y})$, allora vale $C_{x,y} \in L_2(C_{i,j})$. Di conseguenza è corretto dire che tutte le celle in $L_2(C_{x,y})$ hanno nei loro insiemi L_2 la cella $C_{x,y}$. Siccome $|L_2(C_{x,y})| \leq 40$, vale che $C_{x,y}$ è appartenente ad al più 40 insiemi L_2.

Definizione 4.9 (Spessore di $L_1 \cup L_2$)**.** Data una cella $C_{x,y}$ e gli insiemi $L_1(C_{x,y})$ e $L_2(C_{x,y})$. L'insieme $C_{x,y} \cup L_1(C_{x,y}) \cup L_2(C_{x,y})$ è un quadrato di lato 7 celle, dove al centro è localizzato la cella $C_{x,y}$. Lo spessore di $L_1 \cup L_2$ è il minimo numero di celle che viene attraversato da una curva che collega il bordo della cella centrale $C_{x,y}$ al bordo del quadrato. Ne segue che lo spessore di $L_1 \cup L_2$ è 3 celle.

Proprietà 4.10. Se $C_{u,v}$ non è né una L_1 né una L_2-*neighbour* di $C_{x,y}$, e $C_{u,v} \neq C_{x,y}$, allora la distanza tra ogni oggetto $p \in C_{u,v}$ e ogni oggetto $q \in C_{x,y}$ è strettamente maggiore di r.

Dimostrazione. In quanto lo spessore di $L_1 \cup L_2$ è 3 celle, la distanza tra p e q deve per forza superare $3l = \frac{3r}{2\sqrt{2}} > r$. □

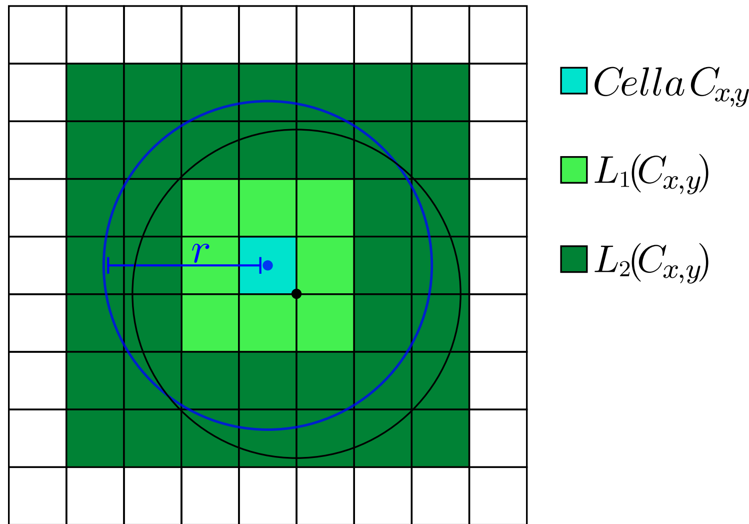


Figura 4.2: Insiemi $L_1(C_{x,y})$ e $L_2(C_{x,y})$

Proprietà 4.11 (Identificazione di outliers basata sulle celle)**.**

a. Se ci sono $> M$ oggetti in $C_{x,y}$, allora *nessun* oggetto in $C_{x,y}$ è un DB(s, r)-outlier.

b. Se ci sono $> M$ oggetti in $C_{x,y} \cup L_1(C_{x,y})$, allora *nessun* oggetto in $C_{x,y}$ è un DB(s, r)-outlier.

c. Se ci sono $\leq M$ oggetti in $C_{x,y} \cup L_1(C_{x,y}) \cup L_2(C_{x,y})$, allora *ogni* oggetto in $C_{x,y}$ è un DB(s, r)-outlier.

La proprietà 4.11-a e 4.11-b sono diretta conseguenza delle proprietà 4.5 e 4.6, la proprietà 4.11-c discende da 4.10.

Nota 4.12. La proprietà 4.11 non copre tutti i casi, infatti esistono delle celle che non soddisfano nessuna ipotesi a-c.
Per tali celle C si è obbligati ridursi a calcolare le distanze tra gli oggetti della cella e quelli di $L_2(C)$, per identificare quali di questi sono DB(s, r)-outlier e quali no.

Ad un livello di astrazione più basso, una cella è un vettore che contiene tutti gli oggetti del dataset che le appartengono. Una cella ha inoltre bisogno di 4 campi addizionali: uno per la coordinata x, uno per la coordinata y, uno per salvare il numero di oggetti interni alla cella (chiamato *count*) e uno per specificare il colore della cella. Una cella può essere etichettata come *rossa* se il numero di oggetti al suo interno è maggiore di $M = N(1 - s)$, come *rosa* se ha meno di M oggetti ma la somma dei suoi oggetti e quelli contenuti nel suo L_1 è maggiore di M, oppure *bianca* se ha meno di M oggetti e la somma dei suoi oggetti e quelli contenuti nel suo L_1 è minore di M. Di default ogni cella è etichettata come *bianca*. Inoltre è necessario dire che le celle sono salvate in un array bidimensionale chiamato *celle*, e per accedere alla cella di riga x e colonna y bisogna usare $celle[x, y]$. Per semplificare la notazione per accedere alla cella $celle[x, y]$ si scriverà $C_{x,y}$.

4.2 Algoritmo

Algoritmo 1: FindAllOutsM(T, N, s, r, R, C)

Input: il dataset T, il numero N di oggetti in T, la frazione s, la distanza r, i numeri R e C di righe e colonne in cui è suddiviso il dataset.

Output: ogni oggetto in T è "segnato" o come DB(s, r)-outlier oppure come NON-DB(s, r)-outlier.

```
1  M ← N(1 − s);
2  for (u = 0; u ≤ R; u++) do
3  |  for (v = 0; v ≤ C; v++) do
4  |  |  C_{u,v}.count ← 0;
5  |  end
6  end
7  for each (p in T)
8  |  (x, y) ← mapToCell(p);
9  |  C_{x,y} ← C_{x,y} ∪ p;
10 |  C_{x,y}.count++;
11 end
12 for (u = 0; u ≤ R; u++) do
13 |  for (v = 0; v ≤ C; v++) do
14 |  |  if (C_{u,v}.count > M) then
15 |  |  |  labelRed(C_{u,v});
16 |  |  end
17 |  end
18 end
19 for each (cella rossa C_{i,j})
20 |  labelPink(L1Neighbours(C_{i,j}));
21 end
22 for each (cella bianca C_{i,j} non vuota)
23 |  Count_{w1} ← C_{i,j}.count + ∑_{C_{h,k} ∈ L_1(C_{i,j})} C_{h,k}.count;
24 |  if (Count_{w1} > M) then
25 |  |  labelPink(C_{i,j});
26 |  else
27 |  |  Count_{w2} ← Count_{w1} + ∑_{C_{h,k} ∈ L_2(C_{i,j})} C_{h,k}.count;
28 |  |  if (Count_{w2} ≤ M) then
29 |  |  |  for each (p in C_{i,j})
30 |  |  |  |  markOutlier(p);
31 |  |  |  end
32 |  |  else
33 |  |  |  for each (p ∈ C_{i,j})
34 |  |  |  |  Count_p ← Count_{w1};
35 |  |  |  |  for each (q ∈ L2Neighbours(C_{i,j}))
36 |  |  |  |  |  if (distance(p, q) ≤ r) then
37 |  |  |  |  |  |  Count_p++;
38 |  |  |  |  |  |  if (Count_p > M) then
39 |  |  |  |  |  |  |  markNotOutlier(p);
40 |  |  |  |  |  |  |  procedi con il prossimo p ∈ C_{i,j} (vai a riga 33);
41 |  |  |  |  |  |  end
42 |  |  |  |  |  end
43 |  |  |  |  end
44 |  |  |  |  markOutlier(p);
45 |  |  |  end
46 |  |  end
47 |  end
48 end
```

In una prima fase l'algoritmo mappa ogni oggetto del dataset nella cella giusta e etichetta come *rossa* ogni cella che contiene $> M$ oggetti, ciò sta a significare che un qualsiasi oggetto in una cella *rossa* non è un DB(s, r)-outlier (vedi proprietà 4.11a). Le celle che sono L_1-*neighbours* di una cella *rossa* vengono etichettate come *rosa*, e quindi (vedi proprietà 4.11b) nessun oggetto in una cella *rosa* può essere un DB(s, r)-outlier.
Le altre celle che soddisfano la proprietà 4.11b sono etichettate come *rosa* nell'**if** di righe 24-25.
Infine, le celle che soddisfano alla proprietà 4.11c sono identificate nell'**if** di righe 28-31, ed ogni oggetto in tali celle è "segnato" come DB(s, r)-outlier.
Si ha mostrato come le proprietà presentate nella Sezione 4.1 siano servite per identificare i DB(s, r)-outlier guardando alle celle piuttosto che ai singoli oggetti, risparmiando così molto tempo.

Come già detto, esistono però delle celle che non soddisfano a nessuna delle proprietà 4.11a-c. In questi casi non si può fare altro che confrontare gli oggetti in tali celle con gli oggetti contenuti nelle celle L_2-*neighbours* (non ha senso guardare oltre, perché gli oggetti sarebbero sicuramente ad una distanza $> r$).
Queste celle sono dette celle *bianche* (C_w), e non appena $count_p$, associato all'oggetto $p \in C_w$, supera M vuol dire che p non può essere un DB(s, r)-outlier e si passa all'oggetto successivo in C_w. Se invece dopo aver calcolato tutte le distanze tra $p \in C_w$ e $q \in L_2(C_w)$, la variabile $count_p$ associato all'oggetto p è $\leq M$, p viene dichiarato DB(s, r)-outlier e si passa all'oggetto successivo in C_w.
Alla fine dell'esecuzione, ogni oggetto in T è "segnato" o come DB(s, r)-outlier oppure è segnato come NON-DB(s, r)-outlier.

L'algoritmo fa uso di otto funzioni ausiliarie.
Le funzioni `distance`(p, q), `markNotOutlier`(p) e `markOutlier`(p) già studiate in precedenza, la funzione `mapToCell`(p) che restituisce in tempo costante la coppia (x, y) (che sono le coordinate della cella giusta per l'oggetto p) con $x = \left\lfloor \frac{x_p}{l} \right\rfloor$ e $y = \left\lfloor \frac{y_p}{l} \right\rfloor$ (dove x_p e y_p sono le coordinate x e y di p). L'algoritmo fa uso anche della funzione `labelRed`$(C_{x,y})$ che setta il campo $C_{x,y}.color = rosso$ e segna come NON-DB(s, r)-outlier tutti gli oggetti interni alla cella, e della funzione `labelPink`$(C_{x,y})$ che, solo se $C_{x,y}.color \neq rosso$, setta il campo $C_{x,y}.color = rosa$ e segna come NON-DB(s, r)-outlier tutti gli oggetti interni alla cella. Queste due funzioni hanno costo lineare rispetto al numero di oggetti appartenenti alla cella $C_{x,y}$. Infine vengono utilizzate le funzioni `L1Neighbours`$(C_{x,y})$ e `L2Neighbours`$(C_{x,y})$ che restituiscono, tramite una lista di coppie di indici (alla coppia (i, j) corrisponde la cella $C_{i,j}$), rispettivamente l'insieme delle celle L_1-*neighbours* e L_2-*neighbours* di $C_{x,y}$. In quanto l'insieme dei *L1-neighbours* di $C_{x,y}$ contiene al massimo 8 celle e l'insieme $C_{x,y}$ contiene al massimo 40 celle, le funzioni hanno quindi costo $O(1)$.

4.3 Analisi di complessità

Work

Per motivi di semplicità l'analisi della complessità sarà discussa per il caso in cui $d = 2$ (sarà poi estesa al caso $d = k$).

- Il primo **for** (righe 2-6) richiede un tempo $O(m)$;
- Il secondo **for** (righe 7-11) richiede un tempo $O(N)$;
- Il terzo **for** (righe 12-18) richiede un tempo $O(m)$;
- In quanto M è il massimo numero di di oggetti che ci possono essere nella circonferenza di raggio r di un outlier, ne segue che ci sono al massimo $\frac{N}{M}$ celle *rosse*. Inoltre la funzione `L1Neighbours`(C) ha costo 8. Quindi il quarto **for** (righe 19-21) richiede un tempo $O\left(8\frac{N}{M}\right) = O\left(\frac{N}{M}\right)$;
- L'analisi della complessità del quinto **for** proposta in [1], procede **ingiustificatamente** affermando che al caso peggiore valgono le seguenti considerazioni:
  i) nessuna cella è stata etichettata né come *rossa* né come *rosa*;
  ii) il numero di oggetti in ogni cella è al massimo M;
  iii) il settimo **for** (righe 33-45) è eseguito su ogni cella;
- Viene ora esposta un'analisi del quinto **for** (righe 22-48) che ha valenza generale. Sia A una cella non vuota che contiene X_A oggetti, A appartiene agli insiemi L_2 di K_A celle *bianche*. Ricordando che ogni cella *bianca* ha al più M elementi, vale che i confronti che vengono fatti rispetto ad A sono $\leq K_A X_A M$. È importante sottolineare che data una cella A il numero K_A è costante e vale che $K_A \leq 40$ (vedi Nota 4.8).
  Per trovare un upper-bound al numero di confronti totali del quinto ciclo, bisogna fare la sommatoria rispetto a tutte le celle A non vuote:

$$\sum_{\substack{\text{celle non} \\ \text{vuote } A}} K_A X_A M \leq 40M \cdot \sum_{\substack{\text{celle non} \\ \text{vuote } A}} X_A \leq 40MN$$

  Dove $\sum_{\substack{\text{celle non} \\ \text{vuote } A}} X_A \leq N$ in quanto la somma degli oggetti appartenenti alle celle non vuote che appartengono agli insiemi L_2 delle celle bianche è al più N. La complessità del quinto **for** è quindi $O(MN)$.

La complessità totale dell'algoritmo è:

$$O(m) + O(N) + O(m) + O(\tfrac{N}{M}) + O(MN) \approx O(m + MN)$$

È importante sottolineare quanto sia conservativa questa analisi. Infatti nella pratica ci si aspetta che siano molte le celle etichettate come *rosse* o *rosa*, riducendo così di molto la complessità del quinto **for**.

4.4 Generalizzazione di FindAllOutsM a dimensioni > 2

Fin'ora è stato presentato ed analizzato l'algoritmo solo nel caso bidimensionale. Viene mostrato ora cosa cambia se ci si sposta in un dataset con dimensione $d > 2$. L'algoritmo FindAllOutsM richiede solo un cambiamento nella *struttura cella* per funzionare anche in d dimensioni. Bisogna modificare quindi la definizione 4.1 di cella:
In quanto la diagonale di un ipercubo in uno spazio d-dimensionale con lunghezza l è $l\sqrt{d}$, per fare in modo che vengano rispettate le proprietà 4.5 e 4.6, deve essere che $l = \frac{r}{2\sqrt{d}}$.

Definizione 4.13 (Cella)**.** Dato un spazio d-dimensionale, una cella è un ipercubo di lunghezza $l = \frac{r}{2\sqrt{d}}$. L'insieme delle celle partiziona l'intero spazio.

Vengono ora fornite le generalizzazioni delle definizioni di L_1 e L_2-*neighbours* presentate nella sezione 4.1.

Definizione 4.14 (L_1-neighbours in d-dimensioni)**.** Data una cella $C_{x_1,\dots,x_d}$ l'insieme L_1-*neighbours* di $C_{x_1,\dots,x_d}$ è:

$$L_1(C_{x_1,\dots,x_d}) = \{C_{u_1,\dots,u_d} \,|\, u_i = x_i \pm 1 \,\forall\, 1 \leq i \leq d, C_{u_1,\dots,u_d} \neq C_{x_1,\dots,x_d}\}$$

Per rispettare la proprietà 4.10, in quanto $l = \frac{r}{l\sqrt{d}}$ il *Layer 2* ha bisogno di essere più spesso rispetto al caso bidimensionale.
Sia x lo spessore del *Layer 2*. Lo spessore complessivo di $L_1 \cup L_2$ è $x + 1$. Affinché la proprietà 4.10 tenga deve essere $l(x + 1) > r$, viene scelta $x = \left\lceil 2\sqrt{d} - 1 \right\rceil$.

Definizione 4.15 (L_2-*neighbours* in d-dimensioni)**.** Data una cella $C_{x_1,\dots,x_d}$ l'insieme L_2-*neighbours* di $C_{x_1,\dots,x_d}$ è:

$$L_2(C_{x_1,\dots,x_d}) = \left\{ C_{u_1,\dots,u_d} \;\middle|\; \begin{array}{l} x_i - \left\lceil 2\sqrt{d} \right\rceil \leq u_i \leq x_i + \left\lceil 2\sqrt{d} \right\rceil \ \forall i \text{ t.c. } 1 \leq i \leq d, \\ C_{u_1,\dots,u_d} \notin L_1(C_{x_1,\dots,x_d}), C_{u_1,\dots,u_d} \neq C_{x_1,\dots,x_d} \end{array} \right\}$$

Lemma 4.16. *Le proprietà 4.5, 4.6, 4.10, 4.11 tengono anche per una struttura cella d-dimensionale.*

Se lo pseudocodice non cambia, la complessità dell'algoritmo invece cambia.
Per i primi quattro cicli la complessità è la stessa del caso $d = 2$, con l'unica differenza che ora m è esponenziale rispetto a d, e non è più necessariamente vero che $m \ll N$. La complessità del quinto **for** (righe 22-48), invece, non è più $O(m+MN)$. Sulla falsa riga di quanto scritto per l'analisi del work nel caso bidimensionale, il numero di distanze calcolate è $\sum_{\substack{\text{celle non}\\ \text{vuote } A}} K_A X_A M$.
La differenza sta nel fatto che K_A non vale più al massimo 40, infatti dipende dalla dimensione del dataset: $K_A \leq (2\left\lceil 2\sqrt{d} \right\rceil + 1)^d - 9$. Quindi il numero di distanze calcolate è $((2\left\lceil 2\sqrt{d} \right\rceil + 1)^d - 9)MN \approx (2\left\lceil 2\sqrt{d} \right\rceil + 1)^d MN$. Sostituendo c a $(2\left\lceil 2\sqrt{d} \right\rceil + 1)$, segue che la complessità dell'algoritmo in d-dimensioni è $O(m + c^d MN)$.

4.5 Considerazioni

Ricapitolando, si ha complessità $O(m + MN)$ nel caso bidimensionale, e complessità $O(m + c^d MN)$ nel caso generale.
Come già sottolineato si osserva che, per valori di M che sono $o(N)$, la dipendenza quadratica rispetto a N sia stata eliminata (si ricorda che gli algoritmi Index-Based e Nested Loop hanno complessità $O(dN^2)$) a scapito dell'introduzione di una dipendenza esponenziale rispetto al numero di dimensioni d.

È opportuno fare una serie di considerazioni:

- se si cercano degli outlier "importanti" è necessario utilizzare un grande valore per r. A un grande valore di r corrispondono un minor numero di celle per ogni dimensione.
- per valori di s vicini ad 1, M è piccolo; si otterranno così un maggior numero di celle *rosse* e *rosa*. Di conseguenza saranno poche le celle *bianche* e quindi il numero di confronti oggetto-oggetto sarà basso.
- all'aumentare delle dimensioni d, il numero di celle vuote aumenterà, si avranno così un numero di celle non vuote molto minore di m.

Viste le suddette considerazioni la reale efficienza di FindAllOutsM è molto più allettante di quanto possa suggerire la complessità mostrata (che appunto era stata calcolata restando molto conservativi).

Un modo per ottenere un algoritmo ancora più veloce è quello di non compiere alcun confronto oggetto-oggetto, ed etichettare gli oggetti solo facendo considerazioni sulle celle. Si otterranno così delle etichette approssimate. Nello specifico si possono eliminare le righe dalla 34 alla 43 dell'algoritmo 4.2 e cambiare la riga 44 usando al posto di `markOutlier`(p) la funzione `markNotOutlier`(p). In questo modo vengono etichettati come NON-DB(s, r)-outlier tutti gli oggetti di una cella *bianca* che ha in L_2 almeno M oggetti.
Si considerino i seguenti casi:

Oggetto $p \in T$	Etichetta corretta	Etichetta approssimata
$p \in$ cella *rossa*	Non Outlier	Non Outlier
$p \in$ cella *rosa*	Non Outlier	Non Outlier
$p \in$ cella *bianca* e L_2 ha $\leq M$ oggetti	Outlier	Outlier
$p \in$ cella *bianca* e L_2 ha $> M$ oggetti	Outlier Non Outlier	Non Outlier Non Outlier

Con l'algoritmo approssimato si sbaglia solo quando un oggetto che è in una cella *bianca*, che ha più di M oggetti in L_2 è in realtà un outlier (si vedrebbe se si facessero i confronti oggetto-oggetto), ma viene identificato come non outlier. In questo caso, però, l'oggetto p ha più di M oggetti a distanza $\frac{3}{2}r$, quindi, nonostante secondo la definizione 2.1 p sia un outlier, si può, facendo un'approssimazione, etichettarlo come non outlier.

Capitolo 5

Algoritmo Solving Set

Fin'ora sono stati presentati tre algoritmi che mirano ad individuare i DB-outliers in un dato dataset piuttosto che sviluppare un modello capace di predire i DB-outliers in "nuovi dati in arrivo".

In questo capitolo, facendo riferimento a [2], si distingue quindi tra *identificazione di outlier* e *predizione di outlier*. Per *identificazione di outlier* si intende l'identificazione dei *top-n outlier* in un dataset T, che sono gli n oggetti con *peso* maggiore (la definizione di *peso* viene data successivamente); mentre per *predizione di outlier* si intende il processo di decidere se un nuovo oggetto in arrivo è un DB-outlier rispetto a T (che vuol dire capire se il suo *peso* in T è $> w^*$, dove w^* è il *peso* del n-simo oggetto con *peso* maggiore in T).
Si introduce il concetto di *Outlier Solving Set* S, che è un sottoinsieme di T, contenente un numero sufficiente di oggetti di T, che permette di considerare solo le distanze tra le coppie $S \times T$ per ottenere i *top n outliers*.
Viene quindi presentato un algoritmo che restituisce il solving set S e ottiene i *top n outlier* in T e il peso w^* evitando di calcolare tutte le ditstanze tra le coppie di oggetti. Inoltre verrà mostrato che il solving set S, oltre a contenere i *top n outlier*, permette di classificare ogni nuovo oggetto come DB-outlier o meno rispetto a T, solo guardando al *peso* dell'oggetto rispetto a S. In questo senso S può essere visto come una rappresentazione compressa di T.

5.1 Definizioni

A differenza degli altri capitoli vengono presentate definizioni diverse (vedi [2]).

Definizione 5.1 (nearest-neighbours)**.** Dato un dataset T, un oggetto p, una funzione `distance` su $T \cup p$, un intero positivo i; allora l'*i-esimo nearest neighbuor* $nn_i(p, T)$ di p rispetto a T è l'oggetto $q \in T$ tale che esistono esattamente $i - 1$ oggetti $v \in T$ (se $p \in T$ va considerato anche p stesso) tali che $\texttt{distance}(p, q) \geq \texttt{distance}(p, v)$.
Vale che se p è $\in T$, allora $nn_1(p, T) = p$; se invece $p \notin T$, $nn_1(p, T)$ è l'oggetto di T più vicino a p.

Definizione 5.2 (peso)**.** Dato un dataset T di N oggetti, una funzione `distance` su T, un oggetto $p \in T$, un intero k con $1 \leq k \leq N$, il *peso* $w_k(p, T)$ di p in T (rispetto a k) è $\sum_{i=1}^{k} \texttt{distance}(p, nn_i(p, T))$.

Intuitivamente il *peso* indica il grado di dissimilarità tra un oggetto e i suoi *nearest-neighbours*, quindi più il *peso* è piccolo più i *nearest-neighbours* sono simili all'oggetto.

Definizione 5.3 ($T_{i,k}$)**.** Si indica con $T_{i,k}$, dove $1 \leq i \leq N$, l'oggetto di T che ha l'*i-esimo peso* più grande rispetto a k in T.
Segue che $w_k(T_{1,k}, T) \geq w_k(T_{2,k}, T) \geq ... \geq w_k(T_{N,k}, T)$.

Definizione 5.4 (Outlier Detection Problem - ODP$\langle T, \texttt{distance}, n, k\rangle$)**.** L'*Outlier Detection Problem* consiste nel trovare gli n oggetti di T che hanno i *pesi* più grandi, rispetto a k, che sono l'insieme $\{T_{1,k}, T_{2,k}, ..., T_{n,k}\}$.
Questo insieme è detto *Solution Set* del problema.

Definizione 5.5 (Outlier Prediction Problem - OPP$\langle T, q, \texttt{distance}, n, k\rangle$)**.** Dato un insieme di oggetti U, il dataset T che è un sottoinsieme di U di dimensione N, un oggetto $q \in U \setminus T$ detto *oggetto di query*, una funzione `distance` su U e due interi n e k, con $1 \leq n, k \leq N$; l'*Outlier Prediction Problem* è definito come segue: Vale $w_k(q, T) \geq w_k(T_{n,k}, T)$? (ovvero chiedersi se q è un outlier rispetto a T).

Nota 5.6. L'ODP può essere risolto in $O(N^2)$, calcolando tutte le distanze $\{\texttt{distance}(p, q) \,|\, (p, q) \in T \times T\}$; mentre il confronto tra l'*oggetto di query* q e tutti gli oggetti in T, con complessità $O(N)$, è sufficiente per risolvere l'OPP.

Viene ora presentato il concetto di *outlier solving set*, e come può essere utilizzato per risolvere sia l'ODP che l'OPP.

Definizione 5.7 (Outlier Solving Set)**.** Dato un dataset T di N oggetti, una funzione `distance` su T e due interi n e k, con $1 \leq n, k \leq N$; un *outlier solving set* per l'ODP$\langle T, \texttt{distance}, n, k\rangle$ è un sottoinsieme S di T tale che:

1. $|S| \geq max\{n, k\}$;
2. Si indichi con $lb(S)$ l'n-esimo elemento più grande di: $\{w_k(p, T) \,|\, p \in S\}$ ($lb(S)$ è un *peso*). Allora, per ogni $q \in (T - S)$, vale $w_k(q, S) < lb(S)$;

Intuitivamente un solving set S è un sottoinsieme di T tale che le distanze $\{\texttt{distance}(p, q) \,|\, p \in S, q \in T\}$ sono sufficienti per dire che S contiene il solution set dell'ODP.

Si indichi con $n^* \geq n$ l'intero positivo tale che $w_k(T_{n,k}, T) = w_k(T_{n^*,k}, T)$, e $w_k(T_{n^*,k}, T) > w_k(T_{n^*+1,k}, T)$.
L'insieme $\{T_{1,k}, ..., T_{n^*,k}\}$ è detto *extended solution set* dell'ODP$\langle T, \texttt{distance}, n, k\rangle$.
La seguente proposizione mostra che $S \supseteq \{T_{1,k}, ..., T_{n^*,k}\}$ e che, quindi, $lb(S)$ è uguale a $w_k(T_{n,k}, T)$.

Proposizione 5.8. *Sia S un solving set per l'ODP$\langle T, \texttt{distance}, n, k\rangle$. Allora $S \supseteq \{T_{1,k}, ..., T_{n^*,k}\}$.*

Dimostrazione. Si procede per assurdo. Si supponga che esista i, $1 \leq i \leq n^*$, tale che $T_{i,k} \notin S$. Allora, $T_{i,k} \in (T - S)$ e $w_k(T_{i,k}, S) \geq w_k(T_{i,k}, T) \geq lb(S)$. Ne segue che S non è un *solving set*, il che è un assurdo. □

Ovviamente T è un solving set e la proposizione mostra che un solving set contiene almeno l'extended solution set $\{T_{1,k}, ..., T_{n^*,k}\}$. In generale un extended solution set non basta a formare un solving set.

Per quanto riguarda l'OPP, viene usato il solving set S (trovato risolvendo l'ODP) per risolvere tale problema, calcolando solo le distanze tra l'*oggetto di query* e gli oggetti in S, piuttosto che con gli oggetti nell'intero dataset T.
In pratica dato un ODP$\langle T, \texttt{distance}, n, k\rangle$, si risolve prima tale problema trovando S, e successivamente, si risponde ad ogni OPP$\langle T, q, \texttt{distance}, n, k\rangle$ nel seguente modo: si risponde "no" se $w_k(q, S) < lb(S)$ (q non è un outlier rispetto a T); si risponde "sì" altrimenti (q è un outlier rispetto a T).

5.2 Algoritmo

Algoritmo 3: SolvingSet(T, n, k, m, g)

Input: il dataset T, il numero n di *top-outliers* da identificare, il numero k di *neighbours*, il numero $m \geq k$ che rappresenta il numero di oggetti da selezionare da T per creare $Cand$ ad ogni iterazione del ciclo while, e il numero razionale $g \in [0, 1]$ che specifica il trade-off tra il numero di oggetti randomici $(m(1-g))$ e il numero di oggetti aventi il *peso* maggiore corrente (gm oggetti) da selezionare per costruire $Cand$.

Output: l'insieme $SolvSet$, che è il solving set del dataset T e l'heap Top che contiene i *top-n outliers* di T.

```
1  SolvSet ← ∅;
2  Top ← ∅;
3  Cand = RandomSelect(T, m);
4  while (Cand ≠ ∅) do
5  |   SolvSet ← SolvSet ∪ Cand;
6  |   T ← T − Cand;
7  |   for each (p in Cand)
8  |   |   for each (q in Cand)
9  |   |   |   δ ← distance(p, q);
10 |   |   |   UpdateMin(NN[p], ⟨q, δ⟩);
11 |   |   |   if (p ≠ q) then
12 |   |   |   |   UpdateMin(NN[q], ⟨p, δ⟩);
13 |   |   |   end
14 |   |   end
15 |   end
16 |   NextCand ← ∅;
17 |   for each (p in T)
18 |   |   for each (q in Cand)
19 |   |   |   if (max{Sum(NN[p]), Sum(NN[q])} ≥ Min(Top)) then
20 |   |   |   |   δ ← distance(p, q);
21 |   |   |   |   UpdateMin(NN[p], ⟨q, δ⟩);
22 |   |   |   |   UpdateMin(NN[q], ⟨p, δ⟩);
23 |   |   |   end
24 |   |   end
25 |   |   UpdateMax(NextCand, ⟨p, Sum(NN[p])⟩);
26 |   end
27 |   for each (q in Cand)
28 |   |   UpdateMax(Top, ⟨q, Sum(NN[q])⟩);
29 |   end
30 |   Cand ← CandSelect(NextCand, T − NextCand, g);
31 end
```

Per risolvere l'ODP l'algoritmo computa i *pesi* degli oggetti del dataset, comparando ogni oggetto p con un piccolo sottoinsieme di T, chiamato $Cand$ (che sta per candidati), e salva i *k-nearest-neighbours*, trovati fino al momento corrente, nell'heap $NN[p]$ (che sta per Nearest Neighbours di p), rispetto a $Cand$. In quanto si stanno calcolando solo le distanze rispetto ad un piccolo sottoinsieme di T, il *peso* corrente di un oggetto è un upper-bound del vero *peso* dell'oggetto (per gli oggetti $q \in Cand$ invece si conosce il vero *peso* di q).
Gli oggetti che hanno *peso* minore del n-esimo *peso* più grande fin'ora calcolato sono detti *non-attivi*, mentre gli altri sono detti *attivi*.

All'inizio $Cand$ contiene oggetti di T scelti randomicamente. Ad ogni iterazione del ciclo $Cand$ è costruito selezionando tra gli oggetti *attivi* di T che non sono stati inseriti in $Cand$ nelle iterazioni precedenti, un mix di oggetti randomici e oggetti che hanno *peso* corrente maggiore (riga 30).

Se durante l'esecuzione un oggetto diventa *non-attivo*, allora non sarà più considerato per la costruzione di $Cand$ in quanto non può essere un outlier. Più avanti si va con l'esecuzione più i *pesi* dei vari oggetti saranno accurati, e quindi ci saranno sempre più oggetti *non-attivi*.

L'algoritmo si ferma quando (condizione a riga 4) non ci sono più oggetti *attivi* che possono essere inseriti in $Cand$, quindi $Cand$ diventa l'insieme vuoto. Il Solving Set è l'unione (riga 5) degli insiemi $Cand$ usati ad ogni iterazione del ciclo.

L'algoritmo fa uso di alcuni heap:

- ad ogni oggetto $p \in T$ è associato un heap $NN[p]$; i cui elementi sono coppie $\langle q, \delta \rangle$, dove q è un oggetto in T e δ è una distanza. Ogni heap memorizza le coppie aventi le k distanze più piccole rispetto a p, ovvero tiene conto di dei *k-nearest-neighbours* di p;
- Top è l'heap di n coppie $\langle p, \sigma \rangle$, dove p sono gli oggetti di T che hanno i *pesi* "veri" maggiori (fino a quel punto dell'esecuzione) e σ è `Sum`$(NN[p])$. Alla fine dell'esecuzione, in Top sono presenti i *top-n outliers* di T;
- $NextCand$ è l'heap di m coppie $\langle p, \sigma \rangle$, dove p sono gli oggetti di T che hanno il più grande upper-bound del *peso*, e σ è `Sum`$(NN[p])$;

L'algoritmo fa uso di sette funzioni ausiliarie:

- la funzione `distance`(p, q) già studiata in precedenza;
- la funzione `RandomSelect`(T, m) seleziona randomicamente m oggetti di T, ha quindi complessità $O(m)$;
- la funzione `UpdateMin`$(NN[p], \langle q, \delta \rangle)$ aggiorna l'heap associato a p:
  - se $NN[p]$ ha meno di k elementi, inserisce la coppia $\langle q, \delta \rangle$.

  - altrimenti sostituisce la coppia $\langle s, \sigma \rangle$, dove σ è la massima distanza in $NN[p]$, con la coppia $\langle q, \delta \rangle$, nel caso in cui $\delta < \sigma$;

  La funzione `UpdateMin` ha costo costante.

- la funzione `Sum`$(NN[p])$ ritorna la somma delle distanze δ dell'heap $NN[p]$ (indica un upper-bound al vero *peso* dell'ogggetto). In quanto il numero di coppie nell'heap è k, la funzione svolge k somme di numeri razionali (le distanze). Il costo della funzione è quindi k;

- la funzione `Min`(Top) ritorna il più piccolo valore σ associato alla coppia memorizzata in Top, ha quindi costo costante;

- la funzione `UpdateMax`$(H, \langle p, \texttt{Sum}(NN[p]) \rangle)$ aggiorna l'heap H:

  - se $H = Top$ e Top ha meno di n elementi, inserisce la coppia $\langle p, \texttt{Sum}(NN[p]) \rangle$.
  - se $H = NextCand$ e $NextCand$ ha meno di m elementi, inserisce la coppia $\langle p, \texttt{Sum}(NN[p]) \rangle$.
  - altrimenti sostituisce la coppia $\langle s, \sigma \rangle$, dove σ è il minimo upper-bound sul *peso* in H, con la coppia $\langle p, \texttt{Sum}(NN[p]) \rangle$ se $\texttt{Sum}(NN[p]) > \sigma$. Al caso peggiore la funzione ha costo $O(k + 2)$ in quanto estrae il massimo di H (costo 1), poi esegue la funzione `Sum`$(NN[p])$ (costo k), e successivamente confronta i due pesi (costo 1);

- la funzione `CandSelect`$(NextCand, T - NextCand, g)$ popola l'insieme $Cand$ per il ciclo successivo. La funzione costruisce un insieme prendendo gm oggetti *attivi* da $NextCand$ e prendendo $(1 - g)m$ oggetti *attivi* in T ma non in $NextCand$. Se non ci sono più oggetti *attivi* la funzione restituisce l'insieme vuoto e l'algoritmo si ferma. In quanto seleziona m oggetti da T, il costo della funzione è semplicemente m;

5.3 Analisi di complessità

Work

L'istruzione a riga 3 ha costo m.
Il ciclo **while** più esterno (righe 4-31) si ferma quando $Cand = \emptyset$, e al caso peggiore ciò accade quando, nelle varie iterazioni del ciclo, è stato caricato in $Cand$ l'intero dataset T. Al caso peggiore tale ciclo compirà $\frac{N}{m}$ iterazioni; e ad ogni iterazione esegue le seguenti istruzioni:
I due cicli **for each** (righe 7-15 e 8-14) compiono ognuno esattamente m iterazioni. Nel ciclo più interno dei due vengono svolte operazioni per un costo totale, al caso peggiore, di $3d + 5$, dove d è la dimensione del dataset. Quindi il blocco di istruzioni che va da riga 7 a riga 15 ha costo: $O(m^2(3d + 5)) \approx O(m^2 3d) \approx O(m^2 d)$.

Il ciclo **for each** (righe 17-26) compie $N - im$ iterazioni, dove i indica a che iterazione del ciclo si è. Ciò accade perchè ad ogni iterazione del ciclo **while** più esterno, alla riga 6, vengono tolti da T gli m oggetti che costituiscono $Cand$.
All'interno di tale ciclo viene eseguito un altro ciclo **for each** (righe 18-24), che compie esattamente m iteerazioni. Ad ogni iterazione di questo ciclo più interno vengono svolte, al caso peggiore, operazioni per un costo totale pari a $2k + 3d + 7$: infatti la condizione dell'**if** a riga 19 ha costo $2k + 3$ e le tre righe successive hanno rispettivamente costo $3d$, 2, 2.
Dopo aver eseguito tale ciclo intero, il ciclo **for each** (righe 17-26) esegue $\texttt{UpdateMax}(NextCand, \langle p, \texttt{Sum}(NN[p])\rangle)$, che ha costo $k+2$. Quindi il blocco di istruzioni che va da riga 17 a riga 26 ha costo: $O((N - im)((k+2) + (m(3d+2k+7)))) \approx O(Nm)$.
L'ultimo ciclo **for each** (righe 27-29) esegue esattamente m iterazioni e ogni iterazione ha costo $k + 2$. Tale ciclo ha complessità $O(m(k + 2)) \approx O(mk)$.
Infine l'ultima istruzione (riga 30) ha costo m.
La complessità totale dell'algoritmo è quindi: $O(m + \frac{N}{m}(m^2 d + Nm + mk + m)) = O(m + N(md + N + k + 1))$ (si noti che è presente il termine N^2).

L'analisi di complessità appena svolta è rigorosa, ma è di difficile comprensione. Se si considerano costose solo le operazioni che mirano al calcolo della distanza tra due oggetti si ottiene un work più intuitivo e snello.

Work Semplificato

È facile vedere che al caso peggiore vengono calcolate tutte le distanze tra le coppie di oggetti in T. La complessità è quindi $O(|T|^2) = O(N^2)$.
Nella pratica però si eseguono solamente i confronti tra gli oggetti di T e gli oggetti di S, ottenendo una complessità pari a $O(|T|^{1+\beta})$, con $\beta < 1$. Infatti dato che S è il solving set trovato dall'algoritmo, il numero di distanze calcolate è $|T|^{1+\beta} = |T| \cdot |S|$, e quindi $\beta = \frac{\log |S|}{\log |T|}$ che è < 1 in quanto $|S| < |T|$.

5.4 Considerazioni

La potenza di questo algoritmo sta nell'identificare i *top-n outliers* di T con complessità meno che quadratica rispetto al numero N di oggetti in T, e (soprattutto) nel fornire il solving set S, che può essere usato per risolvere l'Outlier Prediction Problem (OPP) semplicemente confrontano il nuovo oggetto con i soli oggetti di S, risparmiando così molto tempo.

Come spiegato anche in [2], la predizione di nuovi outlier usando solamente S non è infallibile (infatti non è dimostrato che S basti ad identificare nuovi outlier); ma è stato mostrato sperimentalmente (in [2]) che il numero di falsi-negativi e falsi-positivi (ovvero rispettivamente oggetti che sono oulier ma sono identificati come non outlier e oggetti che non sono outlier ma sono identificati come outlier) è trascurabile.

Bibliografia

[1] Edwin M. Knorr, Raymond T. Ng e Vladimir Tucakov. «Distance-based outliers: algorithms and applications». In: *The VDLB Journal,* vol. 8 (2000), pp. 237–253.

[2] Fabrizio Angiulli, Stefano Bastia e Clara Pizzuti. «Distance-Based Detection and Prediction of Outliers». In: *IEE Trans. Knowledge and Data Eng.,* vol. 18, n. 2 (feb. 2006), pp. 145–160.