Scalable and Cloud Programming - DBSCAN

Giacomo Vallasciani Matricola: 0000954881,
Vincenzo Armandi Matricola: 0001007839 ,
Ilaria Rinaldi Matricola: 0000939972

Giugno 2022

Contents

Introduzione

Il seguente documento ha lo scopo di illustrare il lavoro svolto per il progetto di Scalable and Cloud Computing.
Si è scelto di implementare DBSCAN, un algoritmo di apprendimento automatico non supervisionato, utilizzato per classificare i dati senza etichetta.
La sfida principale dell'utilizzo di questo algoritmo è trovare gli iper-parametri corretti (valori di epsilon e min points) che si adattino all'algoritmo per ottenere dei risultati accurati.
Per ottenere i valori il più accuratamente possibile abbiamo utilizzato l'algoritmo **KNN**, che calcola per ogni partizione del dataset il corretto valore di epsilon.
Il dataset utilizzato è il seguente, il quale contiene dati di misurazioni del consumo di energia elettrica in una famiglia con una frequenza di campionamento di un minuto su un periodo di quasi 4 anni.
Il linguaggio utilizzato per questo progetto è **Scala**, affiancato dal framework per il calcolo distribuito **Apache Spark**.
L'algoritmo proposto è stato eseguito su tipologie di macchine different aventi diverse risorse hardware e su dei cluster creati su Google Cloud Platform (Per consultare le informazioni in dettaglio, vedasi Table 4.1, Table 4.2 e Table 4.3).
Il lavoro svolto, e nel caso specifico il report si articolerà in 5 capitoli:

1. Nel primo capitolo verrà trattato lo stato dell'arte dell'algoritmo DBSCAN e verrà fornita una spiegazione dettagliata del suo funzionamento a livello matematico.
2. Il secondo capitolo verterà sul "preprocessing" dei dati e sulla ricerca degli iper-parametri (epsilon).
3. Nel terzo capitolo sarà spiegato l'approccio utilizzato per l'implementazione di DBSCAN Parallelo.
4. Nel quarto capitolo verrà descritta la distribuzione su Cloud e i risultati ottenuti.
5. Nell'ultimo capitolo verranno tratte le conclusioni e si elencheranno alcuni potenziali sviluppi futuri.

Chapter 1

DBSCAN

La **Clustering Analysis** è una tecnica di apprendimento non supervisionato che partiziona i dati collocandoli in diversi gruppi detti **Cluster**, in relazione ad una o più caratteristiche specifiche, in modo da posizionare all'interno di uno stesso cluster dati simili tra loro secondo le proprietà considerate. La clustering analysis può essere eseguita utilizzando diversi metodi che si differenziano principalmente per il tipo di distanza calcolata. I principali sono:

- **K-Means**: che calcola la distanza tra i punti
- **Affinity Propagation**: utilizzata per calolcare distanze tra grafi
- **Mean-Shift**: anch'esso utilizzato per dati puntiformi
- **DBSCAN**: il quale considera i punti più vicini (nearest points)

Il progetto proposto è incentrato su DBSCAN (Density-Based Spatial Clustering of Applications with Noise), che è un metodo di clustering proposto nel 1996 da Martin Ester, Hans-Peter Kriegel, Jörg Sander e Xiaowei Xu e rientra nell'ambito delle tecniche di clustering. Grazie ad esso è possibile connettere differenti regioni di punti usando come indicatore la densità: aree con densità simili vengono connesse, permettendo cosi di individuare ed isolare gli outlier presenti nell'insieme dei dati. È stata scelta la sopracitata tecnica, in quanto, al contrario di altri algoritmi di clustering su dati puntiformi,ispeziona il dataset tenendo in considerazione anche la densità dei punti presenti al suo interno. Di seguito, in Figure 1.3, possiamo osservare un confronto tra DBSCAN e K-Means, uno degli algoritmi di clustering più diffuso e usato. In Figure 1.2, invece, possiamo notare come i risultati ottenuti da DBSCAN siano molto più pertinenti su dataset contenenti punti molto vicini e quindi molto densi.

	K-means Clustering	DBScan Clustering
1.	Clusters formed are more or less spherical or convex in shape and must have same feature size.	Clusters formed are arbitrary in shape and may not have same feature size.
2.	K-means clustering is sensitive to the number of clusters specified.	Number of clusters need not be specified.
3.	K-means Clustering is more efficient for large datasets.	DBSCan Clustering can not efficiently handle high dimensional datasets.
4.	K-means Clustering does not work well with outliers and noisy datasets.	DBScan clustering efficiently handles outliers and noisy datasets.
5.	In the domain of anomaly detection, this algorithm causes problems as anomalous points will be assigned to the same cluster as "normal" data points.	DBScan algorithm, on the other hand, locates regions of high density that are separated from one another by regions of low density.
6.	It requires one parameter : Number of clusters (**K**)	It requires two parameters : Radius(**R**) and Minimum Points(**M**) R determines a chosen radius such that if it includes enough points within it, it is a dense area. M determines the minimum number of data points required in a neighborhood to be defined as a cluster.
7.	Varying densities of the data points doesn't affect K-means clustering algorithm.	DBScan clustering does not work very well for sparse datasets or for data points with varying density.

Figure 1.1: Confronto tra K-Means e DBSCAN

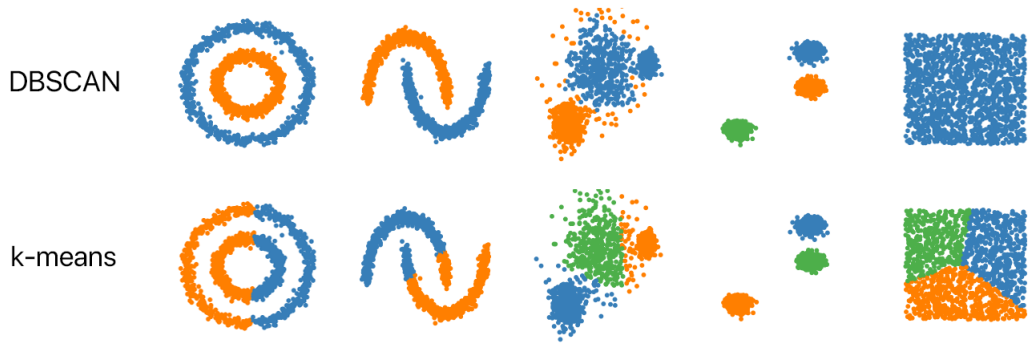


Figure 1.2: Confronto tra i risultati ottenuti da K-Means e DBSCAN

Per capire il funzionamento dell'algoritmo, è necessario introdurre una serie di concetti che saranno poi utili per la sua descrizione:

- **Direttamente Raggiungibile (in densità):** Siano **p** e **q** due punti, **p** è detto direttamente raggiungibile da **q** se essi non sono distanti più di una certa quantità ϵ.
- **Raggiungibilità (in densità):** dati due punti **p** e **q**, si dice che **p** è raggiungibile da **q** se si trova entro una determinata distanza (eps) da esso.
- **Connettività (in densità):** dati due punti **p** e **q**, questi sono connessi (in densità) se esite un terzo punto **z** tale sia **(z,p)** che **(z,q)** siano raggiungibili (in densità).

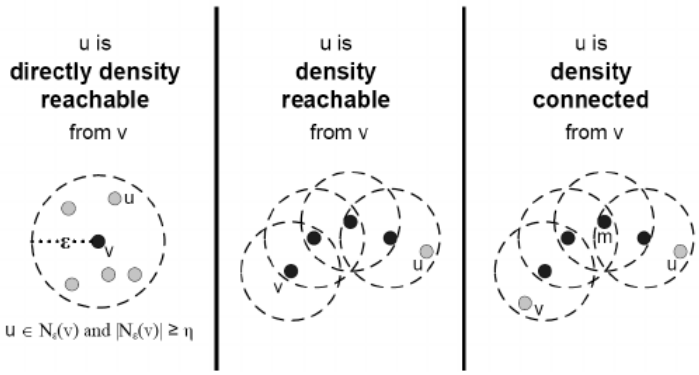


Figure 1.3: Direttamente Raggiungibile, Raggiungibilità e Connettività

- **minPts:** Numero minimo di punti vicini che un determinato punto deve avere per essere considerato un **Core point**
- **Epsilon (ϵ):** Distanza massima tra due punti da considerare come punti vicini (appartenenti allo stesso cluster)
- **Core point:** Punto all'interno di un cluster. Da esso è possibile raggiungere in densità un numero di punti maggiore del numero minimo definito per creare un cluster **(minPts)**
- **Border point:** Punto posto sul bordo di un cluster. Questo è raggiungibile in densità da un **core point** ma, partendo da un border point, non è piu possibile espandere ulteriormente il cluster
- **Noise point:** : Rappresenta un outlier e l'algoritmo non è in grado di collocarlo in alcun cluster con i parametri ϵ e minPts forniti

Funzionamento dell'algoritmo Figure 1.4:

- è selezionato arbitrariamente un punto **p**;
- se **p** è un core point, sono individuati tutti i punti **raggiungibili (in densità)** da esso considerando **Epsilon** e **minPts** ed è formato un **Cluster** che li contiene:
  1. sono aggiunti al cluster appena formato i punti **direttamente raggiungibili (in densità) da p**;
  2. si controlla se tali punti sono a loro volta **core point**;
  3. vengono aggregati i punti a loro volta raggiungibili in densità
- se da **p** è possibile raggiungere un **border point**:
  1. nessun punto è **raggiungibile (in densità) da p**;
  2. si passa ad esaminare un altro punto del dataset non ancora considerato;
- si ripete il procedimento fino alla visita di tutti i punti all'interno del dataset considerato

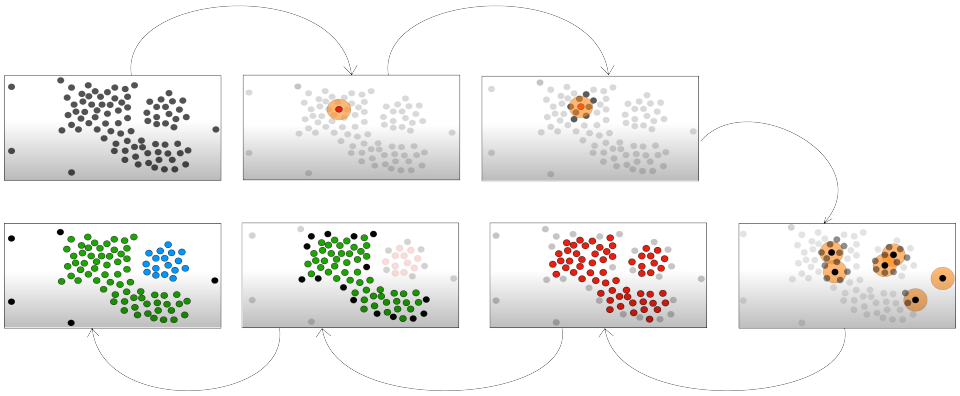


Figure 1.4: Algoritmo DBSCAN Step-by-step

Chapter 2

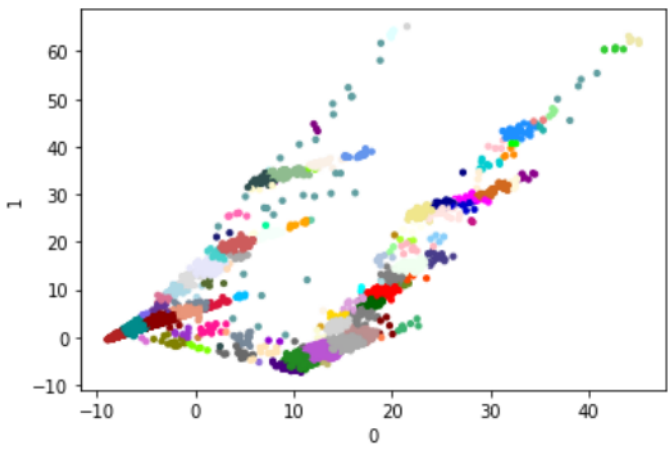
Data Preprocessing e Tuning Parametri

Il dataset scelto per eseguire i test è l'Household Electric Power Consumption, presente su Kaggle.com al seguente link, il quale contiene misurazioni del consumo elettrico di una casa campionate minuto per minuto per 4 anni. Il dataset è composto da più di due milioni di record dei quali è stata selezionata solo una parte in quanto i tempi di esecuzione sulla sua versione completa risultano essere molto lunghi.
Struttura del dataset:

1. **date**: Data in formato dd/mm/yyyy;
2. **time**: Orario in formato hh:mm:ss;
3. **globalactivepower**: Potenza attiva globale della famiglia al minuto (Kw);
4. **globalreactivepower**: Potenza reattiva globale della famiglia al minuto (Kw);
5. **voltage**: Voltaggio al minuto (Volt);
6. **global_intensity**: Intensità globale della corrente al minuto (Ampere);
7. **submetering1**: Sotto-campionamento della cucina (Watt/h di energia attiva);
8. **submetering2**: Sotto-campionamento della lavanderia (Watt/h di energia attiva);

9. **submetering3**: Sotto-campionamento di scaldabagno e climatizzatori (Watt/h di energia attiva);

Inizialmente i dati sono stati pre-processati usando la tecnica denominata Principal Component Analysis o più comunemente nota come **PCA** sulle features dalla numero 3 alla 9, scartando "date" e "time" per ottenere un insieme di punti bidimensionali da dare in input all'algoritmo DBSCAN.
Prima di eseguire l'algoritmo vero e proprio è necessario eseguire prima un sotto-algoritmo (una versione semplificata di KNN che calcola solo le distanze tra punti)attraverso il quale si è stati in grado di determinare il parametro **epsilon** ottimale per ottenere la migliore "clusterizzazione" possibile.
Il secondo parametro richiesto da DBSCAN è **minPts** che serve a determinare il numero minimo di punti per creare un cluster (solitamente settato a $2 \times numFeatures$), nel nostro caso $\mathbf{minPts = 4}$.



	0	1	2
0	15.295269	-1.166883	11
1	16.540941	0.632867	105
2	17.617385	0.663633	105
3	17.384570	0.197421	105
4	14.095957	-1.950366	11
...	...	...	...
24983	9.506471	-6.254688	33
24984	9.390097	-6.331550	45
24985	9.278223	-6.403515	45
24986	9.639342	-6.166587	33
24987	10.797364	-6.087350	33

Figure 2.1: Esempio di clustering di un dataset di 25 mila elementi (a sinistra) e un estratto del dataset pre-processato con i cluster assegnati ad ogni punto (a destra)

Chapter 3

DBSCAN Parallelo

In questo capitolo verrà descritta l'implementazione dell'algoritmo **Scalable-DBSCAN** che utilizza **Apache Sparks**,un framework open-source per l'analisi di grandi quantità di dati su cluster, nato per essere veloce e flessibile. È caratterizzato dalla capacità di memorizzare risultati (solitamente parziali) in memoria centrale e si offre come valida alternativa a Hadoop MapReduce, il quale memorizza i risultati delle computazioni su disco. Ogni programma Spark è strutturato in questo modo (Figure 3.1): si leggono dati da disco in uno o più RDD, li si trasformano e si recupera il risultato della computazione. Le operazioni di trasformazione vengono effettuate solo nel momento del bisogno, cioè quando si richiede un risultato. Spark infatti, per ottenere il contenuto di un RDD, salva in memoria un grafo aciclico diretto (DAG) delle operazioni da eseguire. Le operazioni di trasformazione o di salvataggio/recupero di dati vengono trasformate in una serie di stage eseguiti in sequenza, ognuno dei quali è composto da un insieme di task che vengono eseguiti dagli executor.

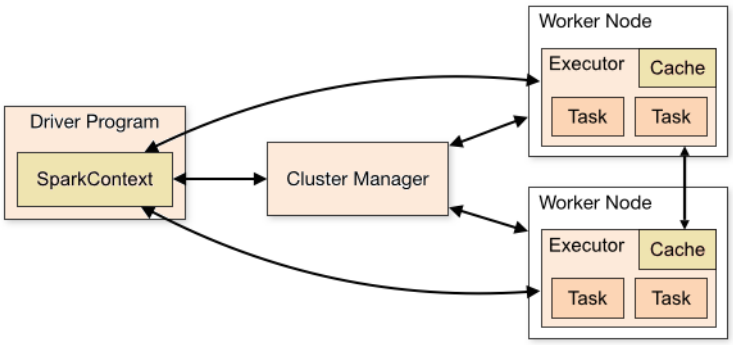


Figure 3.1: Spark overview

Scalable-DBSCAN è un algoritmo sviluppato per consentire il clustering di un gran numero di dati in maniera distribuita. Il primo step consiste nel disporre i dati in rettangoli (**boxes**) bilanciati (che contengono circa la stessa quantità di dati). In seguito, dopo che tutti i dati sono stati disposti all'interno di una box, ognuna di esse viene espansa secondo un certo valore γ, in modo da includere al suo interno tutti i punti che si trovano a distanza γ da essa. A questo punto, l'algoritmo DBSCAN viene eseguito in parallelo su ogni box, nello specifico sui punti che si trovano ai bordi di esse. Una volta esaminati tutti i punti attraverso l'algoritmo, se uno stesso punto è stato etichettato come parte di due cluster diversi, questi ultimi verranno fusi insieme e considerati come un unico gruppo (**Reduce Phase**).
Infine tutti i restanti punti vengono assegnati ai nuovi cluster di appartenza, selezionati a partire da quelli ottenuti nella fase di riduzione.

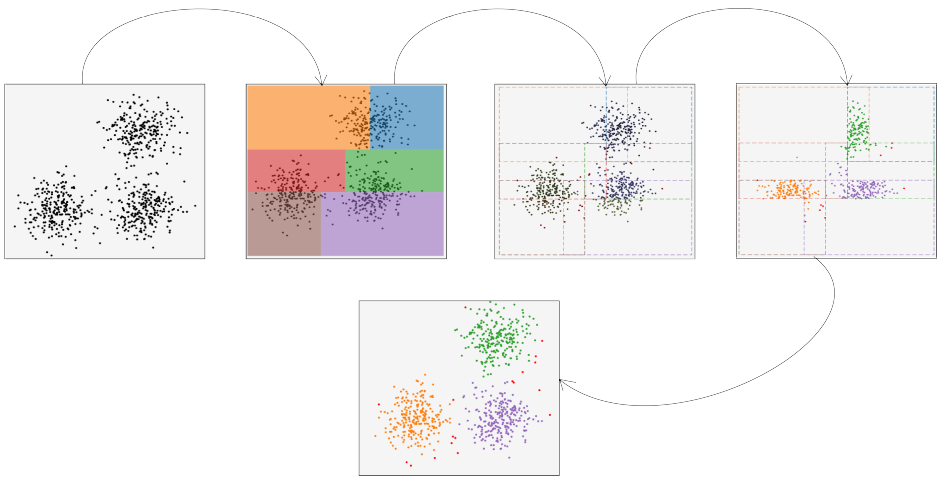


Figure 3.2: Scalable-DBSCAN steps

Il funzionamento di DBSCAN è costituito da tre fasi Figure 3.3:

1. **Data Manipulation:** divide l'intero set di dati in partizioni più piccole in base alla vicinanza spaziale.

2. **Local Clustering:** in questa fase DBSCAN viene eseguito localmente su ogni cluster, il quale genererà dei risulati parziali.

3. **Partial Results Merging and Aggregating:** infine i risultati parziali ottenuti nella fase di clustering vengono aggregati per generare il risultato finale che il sistema produrrà in output.

Nella figura sottostante mostriamo una panoramica del flusso di lavoro di DBSCAN:

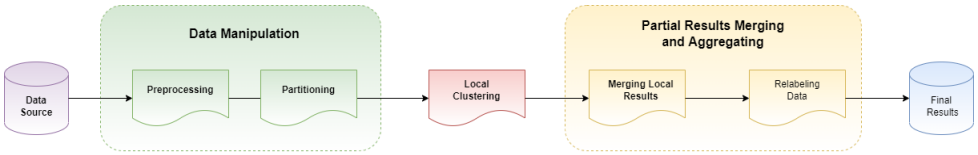


Figure 3.3: Overview di DBSCAN

3.1 Data Manipulation

In questa fase viene eseguito il preprocessing dei dati in modo da normalizzarli ed adattarli alla struttura di input richiesta dall'algoritmo. In seguito, i dati ottenuti dalla fase di preprocessing verranno suddivisi e distribuiti in diverse partizioni per poi essere passati come parametro ai rispettivi cluster incaricati di eseguire l'algoritmo DBSCAN singolarmente (e in parallelo) su essi. Per attuare la suddivisione appena citata (Figure 3.4), si è adottata una strategia denominata **Binary Space Partitioning (BSP)**, attraverso la quale si è stati in grado di creare partizioni del dataset tenendo in considerazione la densità dei dati. Infatti, le partizioni generate conterranno dati considerati vicini in relazione alla prossimità spaziale tra ognuno di essi. Infine, le partizioni generate verrano espanse di un certo valore soglia γ, in modo da inglobare un numero maggiore di punti Figure 3.4.

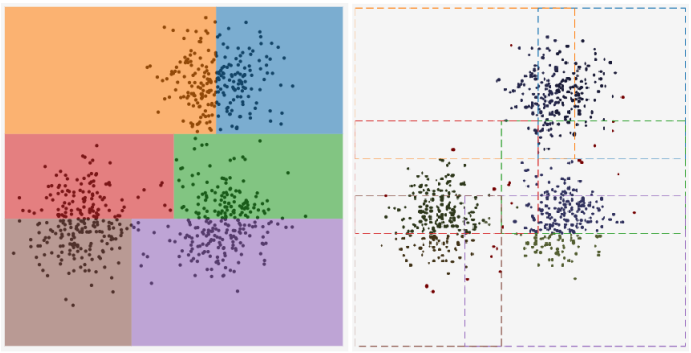


Figure 3.4: ScalableDBSCAN - Partitioning Phase

```
val localPartitions = Partitioner
      .partition(minimumRectanglesWithCount,
   maxPointsPerPartition, minimumRectangleSize)

// assign each point to its proper partition
val duplicated = for {
      point <- data.map(Point)
      ((inner, main, outer), id) <- margins.value
      if outer.contains(point)
    } yield (id, point)
```

Listing 3.1: Estratto di codice per Data Manipulation

3.2 Local Clustering

Nella fase di **Local Clustering**, l'algoritmo DBSCAN viene eseguito in parallelo su ogni partizione generata dalla **Partitioning Phase**, descritta nel dettaglio precedentemente. Durante l'esecuzione dell'algoritmo verranno generati dei cluster relativi ai dati che si trovano in ogni partizione, permettendo ad ogni cluster che ha eseguito l'algoritmo in parallelo di generare dei risultati parziali contenenti la divisione dei punti in gruppi distinti. L'output di ogni singolo cluster sarà una lista di punti, ad ognuno dei quali è assegnato un cluster(gruppo) di appartenenza. Ogni punto all'interno del dataset preso in input, sarà descritto da una coppia (point_coords, cluster_index) la quale associa appunto ogni punto al cluster di appartenenza.

```
// perform local dbscan
val clustered =
      duplicated
        .groupByKey(partitionsNum)
        .flatMapValues(points =>
          new DBSCANCore(eps, minPoints).fit(points))
        .cache()
```

Listing 3.2: Estratto di codice per Local Clustering

3.3 Partial Results Merging and Aggregating

Nell'ultima fase dell'algoritmo, i risultati parziali ottenuti nella fase di clustering vengono riaggregati. Per far ciò, verranno inizialmente considerati solamente i punti presenti nelle intersezioni tra le "Expanded Boxes" generate precedentemente (Figure 3.5, sui quali verrà effettuato un controllo relativo ai cluster di appartenza di ognuno di essi. Se lo stesso punto risultasse essere all'interno di due diversi cluster, questi verranno fusi insieme in uno unico (tramite l'operazione di **reduce**). Infine, una volta individuati i cluster definitivi relativi al dataset iniziale, verrà assegnato il gruppo di appartenenza corretto anche a tutti gli altri punti non ancora considerati (**Relabeling Phase**). L'output finale dell'algoritmo corrisponderà ad una lista di coppie (point_coords, cluster_index), identica a quella descritta nella section 3.2

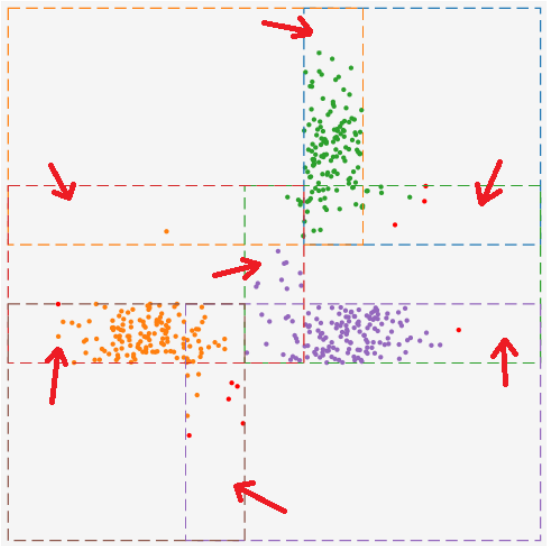


Figure 3.5: ScalableDBSCAN - Merging Phase

```
// generated adjacency graph
val adjacencyGraph = adjacencies.foldLeft(Graph[ClusterId]())
    {
      case (graph, (from, to)) => graph.connect(from, to)
    }

// find all cluster ids
val localClusterIds =
      clustered
        .filter({ case (_, point) => point.flag != Flag.Noise
    })
```

```
10          .mapValues(_.cluster)
11          .distinct()
12          .collect()
13          .toList
14
15  // assign a global Id to all clusters, where connected
    clusters get the same id
16  val (total, clusterIdToGlobalId) = localClusterIds.foldLeft
    ((0, Map[ClusterId, Int]())) {
17      case ((id, map), clusterId) => {
18          map.get(clusterId) match {
19            case None => {
20              val nextId = id + 1
21              val connectedClusters = adjacencyGraph.
    getConnected(clusterId) + clusterId
22              val toadd = connectedClusters.map((_, nextId)).
    toMap(nextId, map ++ toadd)
23            }
24            case Some(x) =>
25              (id, map)
26          }
27        }
28      }
```

Listing 3.3: Estratto di codice per Partial Result Merging and Aggregating

Chapter 4

Cloud e Risultati

Per valutare le performance ottenute dall'implementazione dell'algoritmo Scalable-DBSCAN sono state utilizzate differenti partizioni del dataset principale di diversa cardinalità:

- 10.000 osservazioni
- 25.000 osservazioni
- 50.000 osservazioni
- 75.000 osservazioni
- 100.000 osservazioni
- 150.000 osservazioni

In seguito, per ognuna delle partizione appena descritte, si è eseguito l'algoritmo su macchine con differenti architetture, sia in locale che in Cloud (Google Cloud). Nelle Table 4.1, Table 4.2 e Table 4.3, si possono osservare le informazioni rilevanti relative alle architetture delle macchine utilizzate per condurre i test.

Local Machines			
Machine	CPU	RAM	Storage
Macchina 1	Intel Core i7-7500U CPU 2.90 GHz	16GB DDR4 2.133Mhz	SSD 512GB
Macchina 2	Intel Core i7-12700H CPU 4.70GHz	16GB DDR5 4.800Mhz	SSD 512GB
Macchina 3	Intel Core i5-6200U CPU 2.40GHz	8GB DDR4 2.133Mhz	SSD 512GB

Table 4.1: Local Machines Architecture

Cloud Cluster 1			
Node Type	CPU	RAM	Storage
Master	Intel Haswell - vCpu: 4	15GB	50GB Standard Persistent Disk
Worker 1	Intel Haswell - vCpu: 4	15GB	50GB Standard Persistent Disk
Worker 2	Intel Haswell - vCpu: 4	15GB	50GB Standard Persistent Disk

Table 4.2: Cluster 1 Architecture

Cloud Cluster 2			
Node Type	CPU	RAM	Storage
Master	Intel Cascade Lake e Ice Lake - vCpu: 4	16GB	50GB SSD Persistent Disk
Worker 1	Intel Cascade Lake e Ice Lake - vCpu: 2	8GB	50GB SSD Persistent Disk
Worker 2	Intel Cascade Lake e Ice Lake - vCpu: 2	8GB	50GB SSD Persistent Disk

Table 4.3: Cluster 2 Architecture

Di seguito sono riportati i risultati ottenuti dalle 30 esecuzioni dell'algoritmo sulle diverse architetture e con le differenti partizioni del dataset, raggruppate in relazione alla dimensione del dataset passatogli in input.

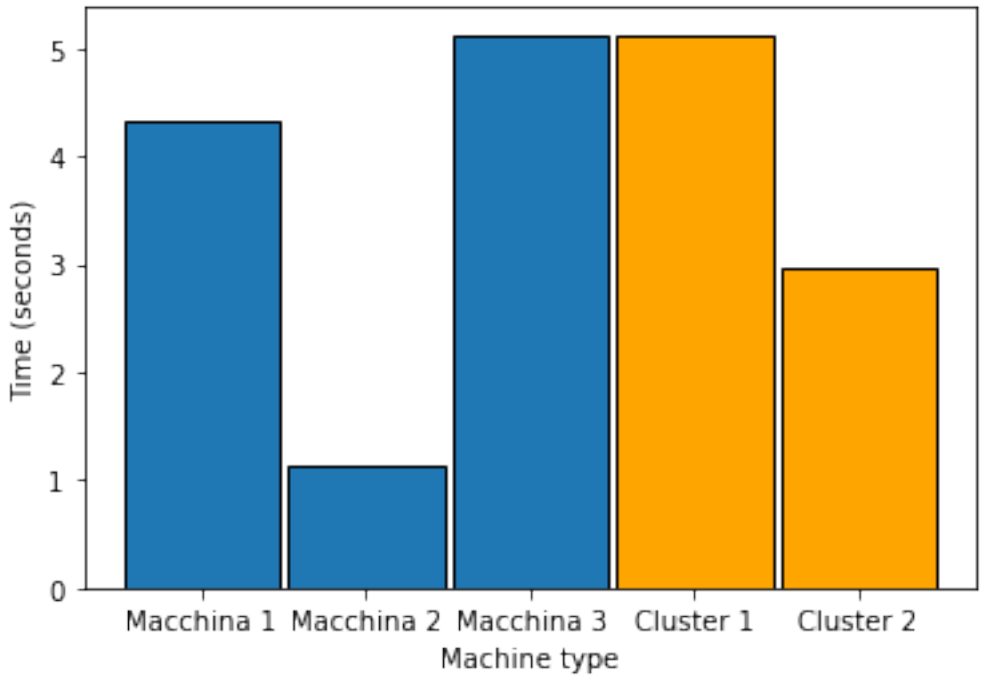


Figure 4.1: Scalable-DBSCAN on 10k dataset

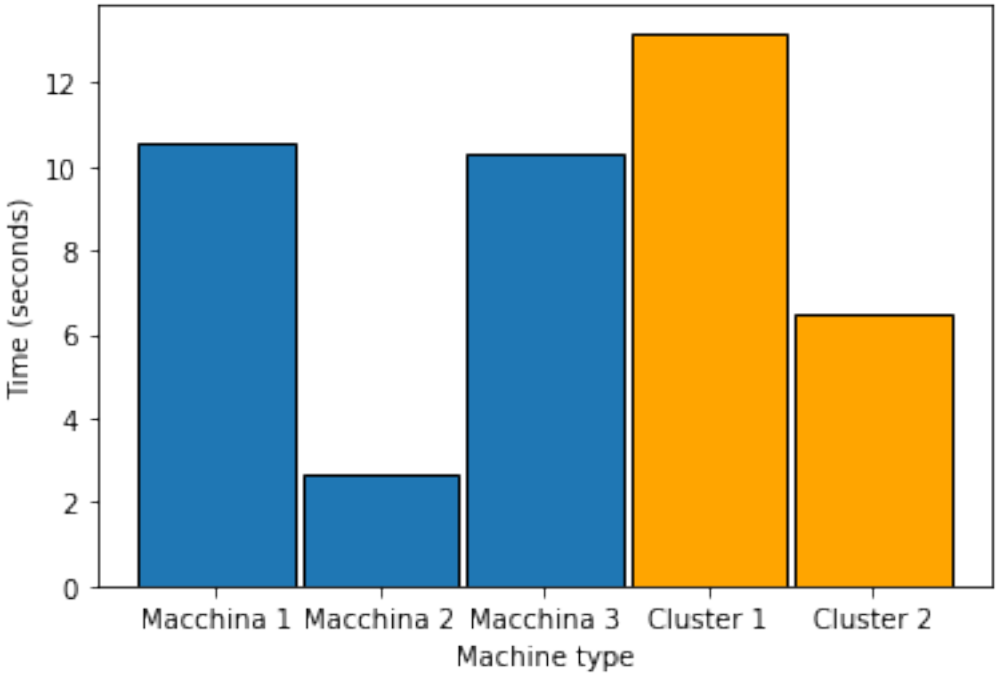


Figure 4.2: Scalable-DBSCAN on 25k dataset

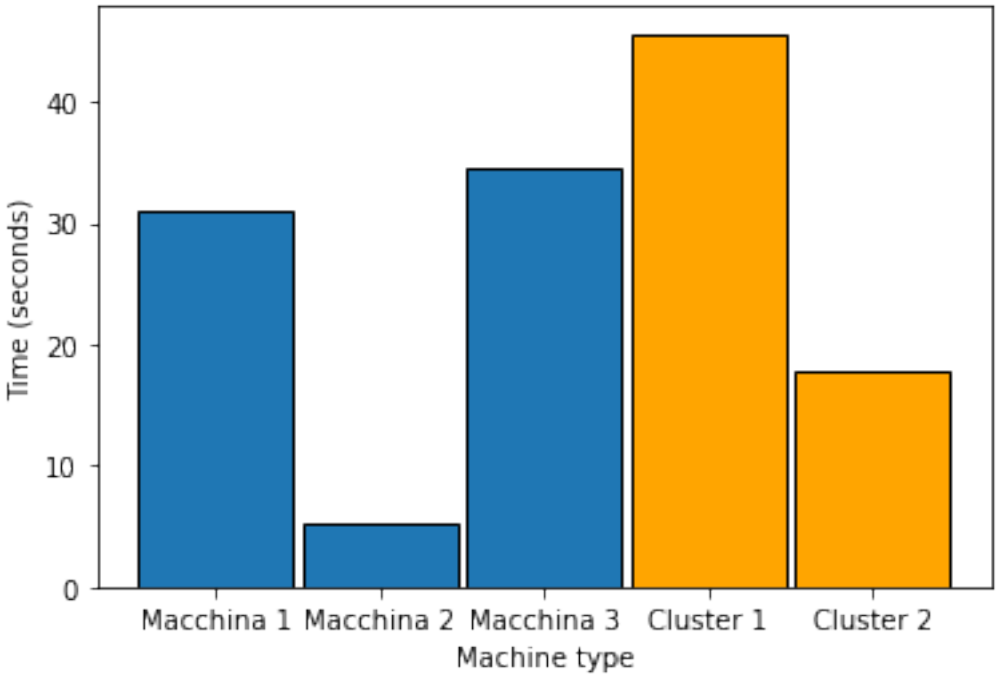


Figure 4.3: Scalable-DBSCAN on 50k dataset

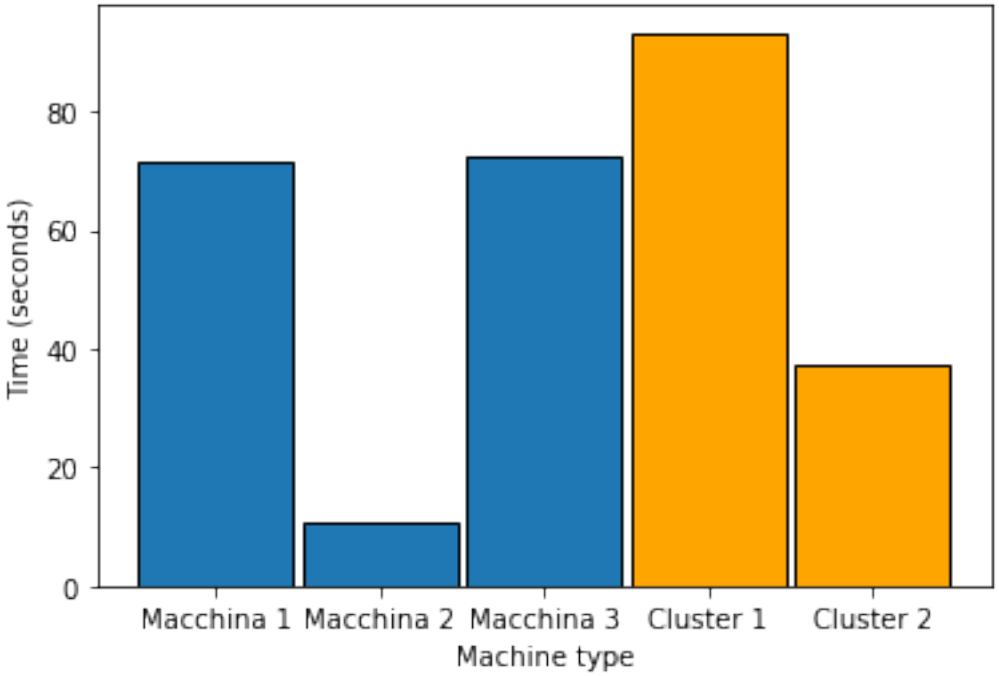


Figure 4.4: Scalable-DBSCAN on 75k dataset

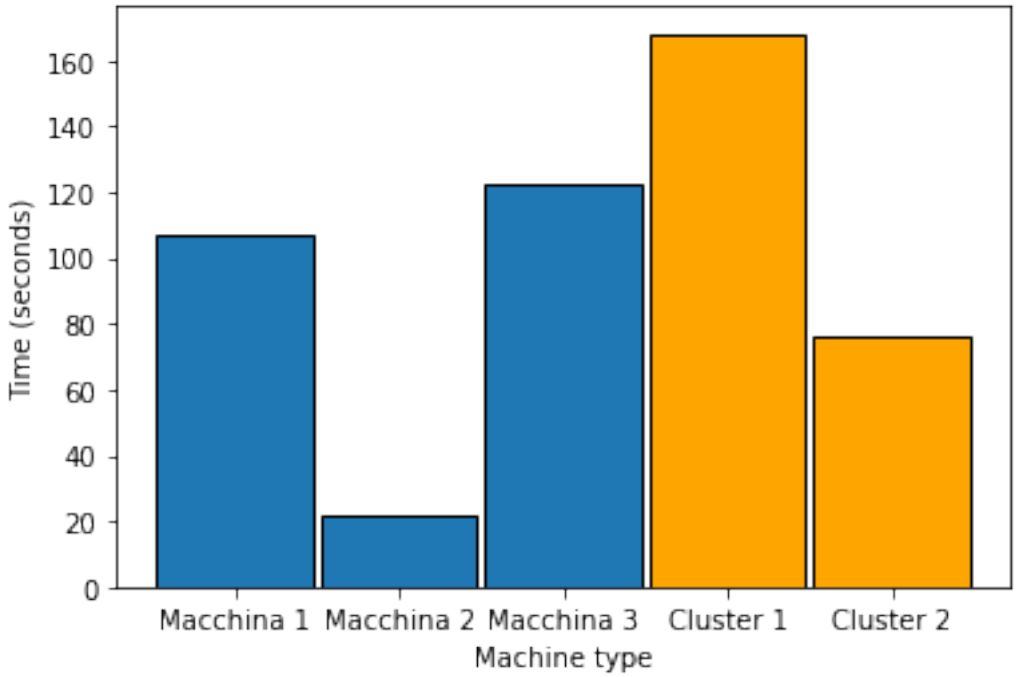


Figure 4.5: Scalable-DBSCAN on 100k dataset

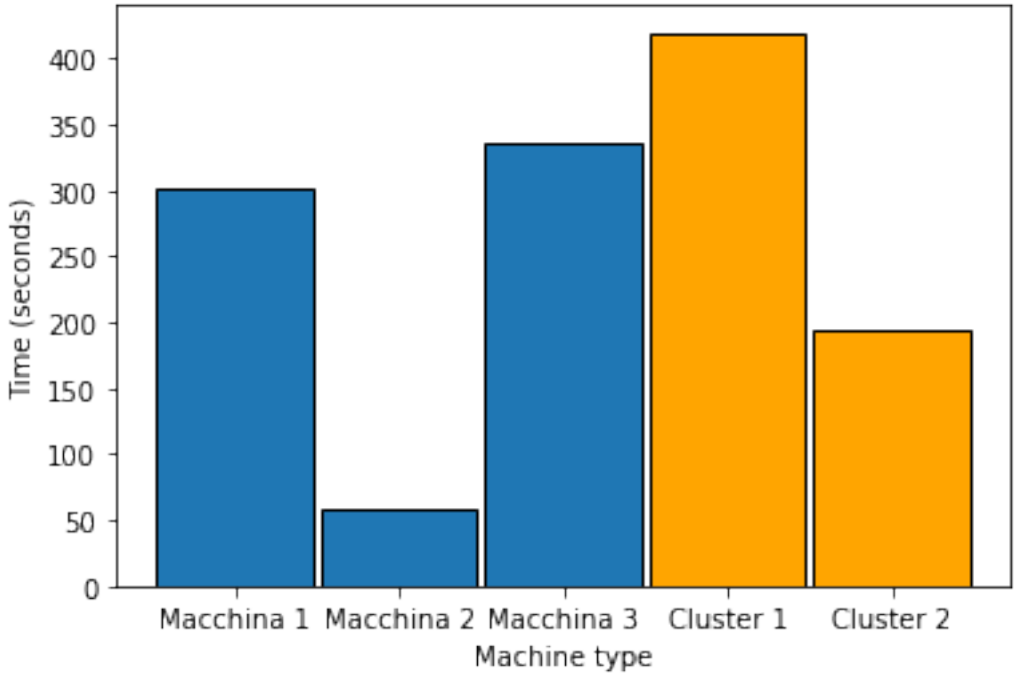


Figure 4.6: Scalable-DBSCAN on 150k dataset

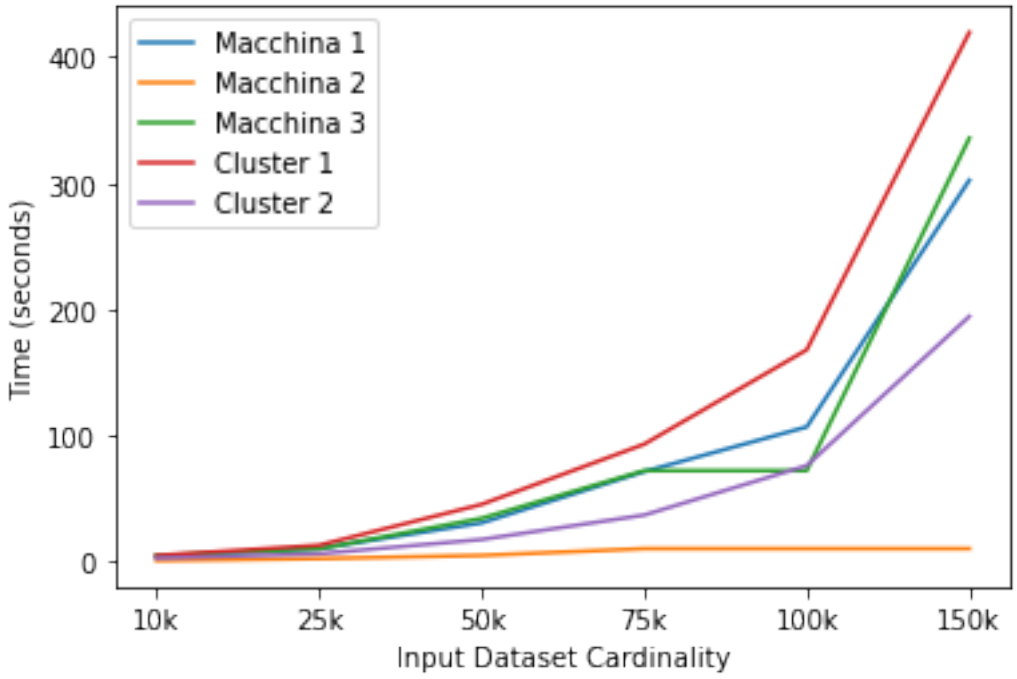


Figure 4.7: Scalable-DBSCAN execution time

Come si evince dai grafici riportati sopra, la "Macchina 2", che è quella con l'hardware migliore ha ottenuto delle performance notevolmente migliori rispetto alle altre 2 macchine. La stessa cosa avviene sui cluster creati su Google Cloud Platform, in quanto il "Cluster 2", al contrario del "Cluster 1" è dotato di dischi SSD ad alte prestazioni e la CPU è di alcune generazioni successive che di conseguenza le permettono di avere una configurazione più performante.

I grafici mostrano inoltre come le performance ottenute sul Cloud non siano del tutto migliori di quelle misurate in locale. Infatti, con tutti i dataset considerati tutte le macchine locali hanno ottenuto performance più elevate rispetto al "Cluster 1", ma solo "Macchina 3" è stata in grado di performare meglio di "Cluster 2". La spiegazione relativa a questo fenomeno si può trovare nel fatto che le configurazioni dei Cluster utilizzano hardware "virtuale" non particolarmente performante dovuto alle limitazioni imposte dall'account di prova di Google Cloud Platform.

In Figure 4.7, possiamo osservare inoltre l'andamento, in termini di tempo, delle istanze dell'algoritmo eseguite sulle 5 architetture considerate con le diverse partizioni del dataset descritte in precedenza.

Chapter 5

Conclusioni

In questo lavoro, è stato presentato l'algoritmo Scalable-DBSCAN, ovvero l'implementazione parallela e distribuita del già noto algoritmo di clusterizzazione DBSCAN.
Durante la fase di progettazione è stata pensata l'architettura del sistema, nello specifico si è cercato di trovare un modo per far si che l'algoritmo tradizionale fosse eseguibile in parallelo e in maniera distribuita. Dopo alcune difficoltà iniziali, si è scelto di partizionare il dataset in **boxes** bilanciate che all'occorrenza durante la fase di **expansion** verranno espanse di un certo offset in modo da garantire il corretto funzionamento del modello proposto (vedasi chapter 3 per la descrizione dettagliata). In seguito, prima di passare alla vera e propria implementazione, tramite il linguaggio **Scala** e il framework **Spark**, è stata necessaria una fase di preprocessing del dataset per normalizzare le osservazioni contenute al suo interno e renderle compatibili con il formato richiesto in input dall'algoritmo. Infine, dopo aver istanziato e lanciato l'algoritmo, si è applicata una fase di post-processing in modo da formattare l'output per permettere di raccogliere i risultati prodotti all'interno di piani cartesiani, così da evidenziare i diversi clustering assegnando ad ognuno di essi un differente colore (Figure 2.1). Dai risultati ottenuti si evince che il sistema sia riuscito ad effettuare l'operazione di clustering in maniera corretta, riconoscendo quindi i gruppi di appartenenza corretti per ciascun elemento, e le sue performance sono nettamente migliorate rispetto all'implementazione di libreria di DBSCAN.
Lavori futuri potrebbero riguardare, la **Partitioning Phase**, in quanto si potrebbe trovare una differente modalità attraverso la quale ripartizionare il dataset, ad esempio ricorrendo a strutture dati più complesse, come alberi e grafi, in modo da abbattere la complessità computazionale della fase di clustering locale dell'algoritmo.

Bibliography

[1] https://towardsdatascience.com/machine-learning-clustering-dbscan-determine-the-optimal-value-for-epsilon-eps-python-example-3100091cfbc

[2] https://medium.com/@mohantysandip/a-step-by-step-approach-to-solve-dbscan-algorithms-by-tuning-its-hyper-parameters-93e693a91289

[3] https://medium.com/@tarammullin/dbscan-parameter-estimation-ff8330e3a3bd

[4] https://blog.sparker0i.me/spark-machine-learning-knn/