Parallelizzazione algoritmi di Clustering con Spark

Edoardo Signoretto VR496214

Agosto 2024

Indice

1 Introduzione

Negli ultimi anni l'ambiente dei Big Data ha avuto una grandissima crescita in termini di ricerca e sviluppo. Quando si parla di Big Data ci si riferisce a insiemi di dati che sono così vasti, complessi e in continua crescita da essere difficili da gestire, analizzare e interpretare utilizzando i tradizionali strumenti di gestione e analisi dei dati. Fra gli ambiti di ricerca del campo dei Big Data si trova il Data Mining. Si tratta del processo di estrazione di informazioni utili e pattern nascosti da grandi quantità di dati grezzi. Una delle tecniche più diffuse è il clustering.

1.1 Clustering

In statistica, il clustering o analisi dei gruppi è un insieme di tecniche di analisi multivariata dei dati volte alla selezione e raggruppamento di elementi omogenei in un insieme di dati. Permette di identificare strutture nascoste all'interno di dati non etichettati, raggruppando elementi simili tra loro in *cluster*. Le tecniche di clustering si basano su misure relative alla somiglianza tra gli elementi. In molti approcci questa similarità è concepita in termini di distanza in uno spazio multidimensionale. La bontà delle analisi ottenute dagli algoritmi di clustering dipende molto dalla scelta della metrica, e quindi da come è calcolata la distanza. Quindi, l'appartenenza o meno a un insieme dipende da quanto l'elemento preso in esame è distante dall'insieme stesso. Gli algoritmi di clustering si suddividono in due grandi gruppi:

- Hard clustering: quando ogni punto di dati appartiene a un solo cluster, come il comune metodo k-means.
- Soft clustering: quando ogni punto di dati può appartenere a più di un cluster, come nel caso dell'algoritmo Gaussian Mixture Models (GMM).

Il clustering è un ambito fondamentale dell'attuale trend dell'intelligenza artificiale. Svolge un ruolo importante in vari domini, offrendo preziose informazioni sui dati e scoprendo modelli e pattern che non sono immediatamente evidenti. In caso di dati non etichettati, in cui la relazione intrinseca tra i punti di dati è nascosta ma necessaria per rivelare informazioni utili, il clustering aiuta a scoprire tali relazioni e a organizzare i dati non etichettati in gruppi significativi. Tuttavia, l'applicazione di algoritmi di clustering su dataset di grandi

dimensioni presenta sfide significative, in particolare in termini di tempo di elaborazione e utilizzo delle risorse computazionali. A tal proposito, per superare questi ostacoli e sfide si ricorre a piattaforme di elaborazione distribuita come Apache Spark.

1.2 Apache Spark

Apache Spark è un sistema di elaborazione open source distribuito, utilizzato in genere con i carichi di lavoro per i Big Data. In particolare, fornisce API di sviluppo in diversi linguaggi di programmazione e supporta il riutilizzo del codice su più carichi di lavoro. Le operazioni supportate sono diverse, fra queste: elaborazione in batch, query interattive, analisi in tempo reale, machine learning ed elaborazione di grafici. Apache Spark offre una soluzione potente e scalabile di parallelizzazione dei processi di clustering, riducendo così significativamente i tempi di elaborazione e permettendo di lavorare con dataset di dimensioni elevate.

1.2.1 Ecosistema Apache Spark

Come mostrato nella figura 1 l'ecosistema Spark comprende cinque componenti chiave:

- Spark Core è un motore di trattamento dati distribuito e per uso generico. Comprende librerie per SQL, l'elaborazione dei flussi, il machine learning e il calcolo dei grafici: tutti elementi che saranno utilizzati nel progetto in esame.
- Spark SQL è il modulo Spark per lavorare con dati strutturati. Consente di eseguire query di dati strutturati all'interno dei programmi Spark, utilizzando SQL o un'API DataFrame familiare.
- Spark Streaming semplifica la creazione di soluzioni per flussi di dati scalabili e la tolleranza di errore. Utilizza l'API integrata nel linguaggio per l'elaborazione dei flussi e supporta Java, Scala e Python.
- MLlib è la libreria Spark scalabile per il machine learning con strumenti che rendono lo sviluppo più facile. MLlib contiene molti algoritmi di apprendimento comuni, come la classificazione, la regressione, i suggerimenti e il clustering.
- GraphX è l'API Spark per i grafici e il calcolo grafico parallelo. È flessibile e funziona perfettamente sia con i grafici che con le raccolte.

1.2.2 Vantaggi di Apache Spark

Spark presenta diversi vantaggi che hanno portato alla scelta di questa tecnologia per il progetto in esame:

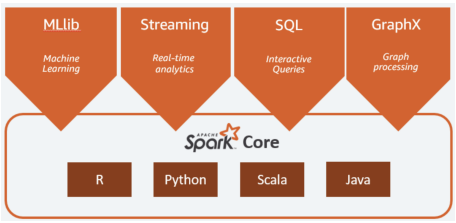


Figura 1: Ecosistema di Spark

- Rapidità: Spark può eseguire query analitiche rapide su dati di qualsiasi dimensione, utilizzando la cache in memoria e l'esecuzione ottimizzata delle query.
- Ideale per sviluppatori: Apache Spark offre supporto nativo per Python e Java attraverso API che semplificano il lavoro e riducono la quantità di codice.
- Carichi di lavoro multipli: Apache Spark è in grado di eseguire carichi di lavoro multipli, tra cui query interattive, analisi in tempo reale, machine learning ed elaborazione di grafici.

Per i motivi appena descritti il codice sviluppato in questo progetto è scritto con linguaggio Python.

1.2.3 Resilient Distributed Datasets (RDD)

Gli Resilient Distributed Datasets sono strutture di dati fondamentali in Apache Spark, progettate per essere collezioni partizionate, distribuite e consapevoli della località. Queste collezioni sono distribuite tra i nodi di un cluster e possono essere memorizzate nella memoria principale, quando le risorse lo consentono, oppure su disco locale se la memoria non è sufficiente. Gli RDD puntano a una fonte di dati diretta, come ad esempio HDFS (Hadoop Distributed File System), e sono in grado di applicare trasformazioni su altri RDD per generare nuovi elementi di dati. Questa caratteristica consente di creare pipeline di elaborazione dei dati che possono essere facilmente composte attraverso operatori avanzati. L'obiettivo principale di Spark è quello di supportare una vasta gamma di operatori, andando oltre le semplici operazioni di Map e Reduce.

- Gli RDD possono essere trasformati usando operatori come map, filter, join, e molte altre trasformazioni, offrendo una grande flessibilità nel trattare i dati.
- Gli algoritmi manipolano i dati attraverso RDD che permettono l'esecuzione parallela e includono sia trasformazioni che azioni.
- Le trasformazioni come map e filter permettono di creare nuovi RDD applicando funzioni ai dati, mentre le azioni come contare, raccogliere o

salvare producono risultati effettivi, come la raccolta dei dati elaborati o la loro scrittura su disco.

Una caratteristica importante degli RDD è la loro resilienza. In caso di guasto di una macchina, gli RDD possono essere ricostruiti automaticamente grazie alla registrazione delle dipendenze dai loro RDD genitori. Questo meccanismo di recupero garantisce che i dati possano essere ripristinati senza perdita di informazioni, anche in ambienti distribuiti su larga scala.

1.3 Dataset

Il dataset utilizzato appartiene al progetto UC Irvine Machine Learning Repository. Una raccolta di database, teorie di dominio e generatori di dati utilizzati dalla comunità dell'apprendimento automatico per l'analisi empirica degli algoritmi. Nello specifico, Individual Household Electric Power Consumption, è un insieme di misurazioni del consumo di energia elettrica in una famiglia con una frequenza di campionamento di un minuto per un periodo di quasi 4 anni. Un insieme di 2.075.259 misure raccolte in una casa situata a Sceaux (7 km da Parigi, Francia) nel periodo fra Dicembre 2006 e Novembre 2010 per un totale di 47 mesi. L'immagine 2 mostra la posizione geografica.

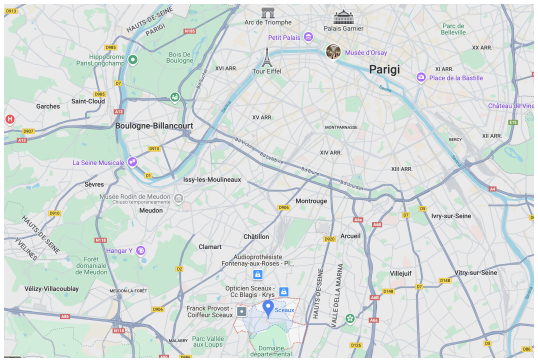


Figura 2: Mappa di Sceaux

Si sottolinea che il dataset contiene alcuni valori mancanti nelle misurazioni (quasi l'1,25% delle righe). Questi sono stati gestisti con tecniche di pulizia di dati spiegate nei successivi paragrafi. Le colonne presenti nell'insieme dei dati, invece, sono le seguenti:

- **Date**: data nel formato dd/mm/yyyy
- **Time**: orario nel formato hh:mm:ss
- **Global_active_power**: potenza attiva globale media al minuto della casa (in kilowatt)

- **Global_reactive_power**: potenza reattiva globale media al minuto della casa (in kilowatt)
- **Voltage**: tensione media al minuto (in volt)
- **Global_intensity**: intensità di corrente globale media al minuto della casa (in ampere)
- **Sub_metering_1**: sotto-misurazione energetica 1 (in wattora di energia attiva). Corrisponde alla cucina, che contiene principalmente una lavastoviglie, un forno e un microonde (i fornelli sono a gas e non elettrici).
- **Sub_metering_2**: sotto-misurazione energetica 2 (in wattora di energia attiva). Corrisponde alla lavanderia, che contiene una lavatrice, un'asciugatrice, un frigorifero e una luce.
- **Sub_metering_3**: sotto-misurazione energetica 3 (in wattora di energia attiva). Corrisponde a uno scaldabagno elettrico e un condizionatore d'aria.

La scelta del dataset in questione è dettata dal numero di dati e dall'insieme delle diverse variabili a disposizione che garantiscono ampia possibilità di applicare tecniche di analisi dei dati. La grossa molte di dati ha permesso di applicare la parallelizzazione agli algoritmi per confrontare l'esecuzione parallela con quella standard.

1.4 Strumenti utilizzati

Il progetto è stato svolto con i seguenti strumenti:

- Macbook Air M2
- 8GB di RAM
- Visual Studio Code + Jupyter Notebook
- Python 3.9.6
- Spark 3.5.1
- Librerie Python: pandas, sklearn, matplotlib, seaborn, pyspark

1.5 Tipologie di clustering implementate

Con il clustering applicato all'insieme di dati in questione è possibile ottenere informazioni importanti come profili di consumo energetico, identificazione di anomalie o malfunzionamenti e segmentazione dei dispositivi in base ai pattern di consumo. Questa analisi può essere utile per migliorare l'efficienza energetica o monitorare l'infrastruttura elettrica. In particolare il clustering può aiutarti a raggruppare le istanze in base a livelli simili di consumo energetico. Nel

contesto del rilevamento di anomalie, invece, cluster con combinazioni anomale di potenza, tensione e intensità possono indicare un problema nell'infrastruttura elettrica. Gli outlier rilevati nei cluster potrebbero segnalare malfunzionamenti di apparecchiature elettriche o consumi anomali in specifiche condizioni. In questo progetto si sviluppano i seguenti algoritmi di clustering:

- KMEANS
- DBSCAN (Density-Based Spatial Clustering of Applications with Noise)
- GMM (Gaussian Mixture Model)

Si è scelto di studiare questi algoritmi in quanto rappresentano tre approcci distinti e complementari all'analisi dei dati: KMEANS si basa su centroidi e minimizzazione della varianza interna ai cluster, DBSCAN utilizza la densità dei punti per identificare gruppi e rumore, mentre GMM adotta un modello probabilistico per descrivere i dati come una combinazione di distribuzioni gaussiane. Questa scelta permette di esplorare il parallelismo in contesti differenti, evidenziando come le caratteristiche specifiche di ciascun metodo influenzino la progettazione e l'implementazione di soluzioni parallele per migliorare efficienza e l'accuratezza.

2 KMEANS

2.1 Introduzione dell'algoritmo

Il K-Means è uno degli algoritmi di clustering più semplici ed efficaci nell'ambito dell'apprendimento non supervisionato. La sua principale funzione è quella di suddividere un insieme di dati in K cluster basandosi sulle caratteristiche comuni dei dati, dove K è un parametro scelto dall'utente che rappresenta il numero di cluster desiderati. L'algoritmo può essere descritto con 5 step:

- Gli elementi del dataset vengono assegnati casualmente ai K cluster definiti inizialmente.
- Per ogni elemento viene calcolata la distanza tra esso e tutti i centroidi, che rappresentano i punti medi dei cluster.
- Ogni elemento del campione viene assegnato al cluster il cui centroide è più vicino.
- I centroidi vengono ricalcolati in base ai punti assegnati a ciascun cluster, rappresentando nuovi punti medi.
- Il processo di assegnazione e ricalcolo dei centroidi viene ripetuto fino a quando non si verifica la convergenza.

2.2 Scelte implementative

K è un parametro in input al K-Means e il suo valore può cambiare completamente il risultato del clustering. Non esiste un metodo assoluto per determinare il numero di cluster e non esiste un'unica soluzione ma esistono diverse tecniche per studiare le soluzioni migliori. Si è scelto di applicare il *metodo del gomito*. Il metodo consiste nel valutare la somma degli errori al quadrato (SSE) per un intervallo di valori di K compreso tra 1 e 10, individuando il punto in cui l'incremento della riduzione dell'errore inizia a diminuire significativamente, formando un angolo (il "gomito"). Questo punto suggerisce il numero di cluster ottimale. Di seguito si riporta il codice Python utilizzato per implementare questa tecnica:

```
sse = []
k_range = range(1, 11)

for k in k_range:
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(scaled_data)
    sse.append(kmeans.inertia_)
```

Nel codice, la variabile *sse* viene utilizzata per memorizzare l'inerzia (SSE) calcolata per ogni valore di K. Successivamente, il risultato è stato rappresentato graficamente, come mostrato nella figura 3.

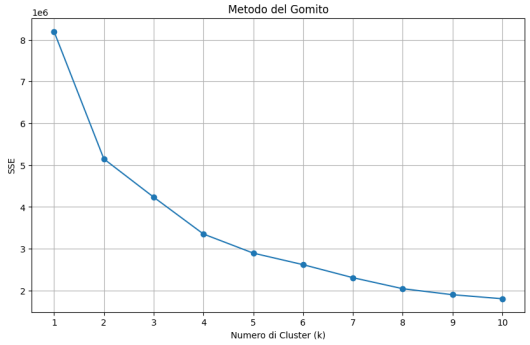


Figura 3: risultato metodo del gomitolo

Il grafico evidenzia il punto in cui l'aggiunta di ulteriori cluster non porta a una riduzione significativa dell'errore, facilitando così la scelta del valore di K. Si è scelto di utilizzare $K = 3$ per analizzare l'algoritmo. Un'altra soluzione poteva essere quella di $K = 4$. Nel contesto dell'algoritmo K-means, un altro parametro che viene settato dall'utente è il parametro *max_iters*. Questo rappresenta il numero massimo di iterazioni che l'algoritmo esegue durante il processo di ottimizzazione. Questo parametro serve come un criterio di arresto per prevenire che l'algoritmo continui a iterare. Nel progetto si è scelto di

utilizzare un valore di $max\_iters = 10$ per confrontare l'implementazione standard con quella parallela. Si esegue comunque l'algoritmo in entrambi i contesti con parametri differenti per confrontarne i risultati. L'algoritmo di K-Means si applica selezionando le seguenti colonne: *Global active power, Global reactive power, Voltage, Global intensity*. Queste variabili possono essere utilizzate, ad esempio, per l'identificazione di classi di consumo energetico.

2.3 Implementazione standard

Per analizzare la versione standard dell'algoritmo, sono state sviluppate due implementazioni distinte. La prima utilizza la libreria scikit-learn, la seconda invece è stata realizzata interamente da zero, senza l'ausilio di librerie predefinite. Questa scelta è stata dettata dal fatto che l'ausilio della libreria ottimizza l'esecuzione in maniera automatica, permettendo di ottenere tempi e performance adeguate. Infatti di seguito viene riportata una media delle performance dell'applicazione dell'algoritmo con libreria:

- Tempo di esecuzione: 3 *minuti*
- Memoria utilizzata: $-16.5469\ MiB$

Si osserva dunque che l'esecuzione è molta veloce e l'utilizzo di memoria si presenta con un valore negativo. Questo perché il sistema operativo ottimizza la memoria allocata ridistribuendola o liberandola mentre il programma è in esecuzione. In figura 4 si mostra il risultato del clustering, in particolare si mostrano 15 punti casuali assegnati ai 3 cluster generati.

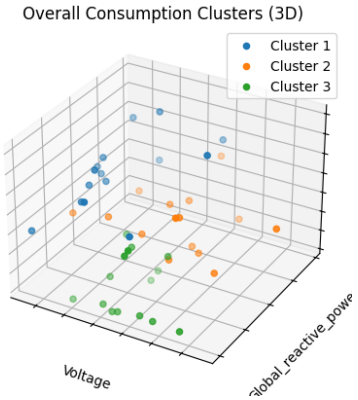


Figura 4: Implementazione standard kmeans con libreria

Implementazione "pura" dell'algoritmo KMEANS si è scelto di strutturarla con le seguenti funzioni:

- *euclidean_distance* calcola la distanza euclidea fra due vettori

- *initialize_centroids* inizializza casualmente K centroidi
- *assign_clusters* assegna ogni punto al centroide più vicino
- *update_centroids* aggiorna i centroidi calcolando la media di tutti i punti assegnati a ciascun cluster

Infine la funzione *kmeans_function* mostrata nel codice seguente richiama le funzione descritte in precedenza per eseguire l'algoritmo:

```
def kmeans_function(X, k, max_iters=10):
    centroids = initialize_centroids(X, k)

    for _ in range(max_iters):
        clusters = assign_clusters(X, centroids)
        new_centroids = update_centroids(X, clusters, k)
        if np.all(centroids == new_centroids):
            break
        centroids = new_centroids

    return centroids, clusters
```

Si osserva infatti che l'algoritmo termina se si raggiunge il massimo numero di iterazioni, passato come parametro, o se i centroidi non cambiano rispetto allo step precedente (cioè si è raggiunga la convergenza). La figura 5 mostra il risultato dell'algoritmo nella sua versione senza ausilio della libreria. Si può affermare che l'output delle due versioni è simile ma differiscono per in termini di performance.

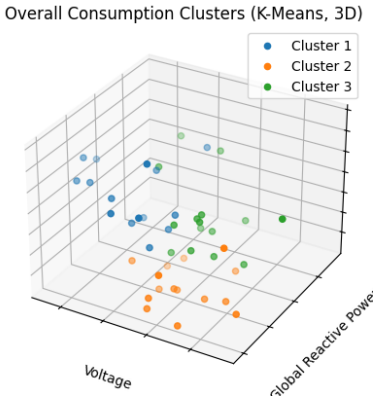


Figura 5: Implementazione standard kmeans

Il tempo medio di esecuzione dell'implementazione senza ausilio della libreria è di circa 16 minuti mentre per il codice con supporto di librerie è circa 3 minuti. La differenza fra i due è dovuta principalmente all'ottimizzazione che il codice

di libreria esegue in automatico e a causa dell'elevato numero di righe presenti nel dataset. I tempi di esecuzione e il consumo delle risorse mostrano in media gli stessi rapporti modificando anche il numero di cluster richiesti. L'immagine 6 mostra il risultato del clustering con un valore di $k = 4$.

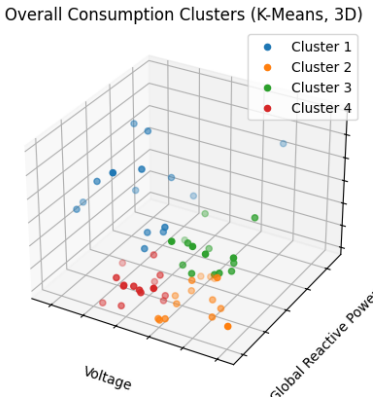


Figura 6: risultato con k = 4

Modificando invece il valore di iterazioni massime il tempo di esecuzione cresce in maniera esponenziale. Infatti, applicando kmeans puro con $k = 3$ e $max\_iter = 100$ il tempo di esecuzione è in media 24 minuti. Per ottimizzare le prestazioni e ridurre i tempi di elaborazione, dunque, la soluzione adottata è l'implementazione della parallelizzazione utilizzando Apache Spark. Questa tecnologia consente di distribuire il carico di lavoro su più nodi di calcolo, sfruttando la potenza di elaborazione parallela per gestire dataset di grandi dimensioni in modo più efficiente.

2.4 Implementazione parallela

L'idea di base di K-Means distribuito è memorizzare i dati in un RDD. Un RDD, o *Resilient Distributed Dataset*, introdotta precedentemente è la struttura dati fondamentale in Spark che consente l'elaborazione parallela. Nel codice seguente è mostrato come si ottiene la memorizzazione dei dati in un RDD:

```
rdd = selected_df.rdd.
    map(lambda row: Vectors.dense(row))
```

Il codice converte il DataFrame *selected_df* in una RDD e poi trasforma ciascuna riga in un vettore denso usando *Vectors.dense*. Per applicare la parallelizzazione è necessario manipolare l'array dei centroidi. Nello specifico ogni worker deve avere una copia del vettore dei centroidi per essere indipendente dagli altri per calcolare l'assegnazione dei cluster. Successivamente è necessario ricalcolare i nuovi centroidi in base all'assegnazione. Chiaramente, si devono mischiare i dati sulla rete in modo da avere, per ogni centroide corrente, l'elenco di tutti i punti dati che gli sono stati assegnati.

```
datapoints = ...
centroids = ...

for itr in range(maxiter):
    bcCentroids = sc.broadcast(centroids)

    closest = datapoints.mapPartition(assignement_step)
    centroids = closest.reduceByKey(update_step_sum).
        map(update_step_mean).collect()
```

Il comando *broadcast* è utilizzato per condividere variabili tra tutti i nodi di un cluster in modo efficiente. In particolare si condivide la RDD creata ad ogni task in maniera indipendente. In questo pseudocodice si definiscono 3 step fondamentali:

- Definizione dei dati e dei centroidi: i dati vengono definiti e inizialmente vengono scelti in modo casuale i centroidi.
- Ciclo di iterazione dell'algoritmo: ad ogni iterazione, i centroidi correnti vengono trasmessi a tutti i nodi del cluster utilizzando le variabili di broadcast. Il passo di assegnazione viene eseguito in parallelo per mappare i punti in partizioni.
- Aggiornamento dei centroidi: i dati vengono ridotti in parallelo utilizzando la funzione reduceByKey per aggiornare i centroidi. Si applica la mappatura per calcolare la media dei nuovi centroidi, che vengono poi raccolti nel driver.

```
for itr in range(maxiter):

    closest = normalized_rdd.
        mapPartitions(assignement_step)

    centroids = closest.
        reduceByKey(update_step_sum).
        mapValues(update_step_mean).collect()

    centroids = [centroid for _,
        centroid in centroids]

    bcCentroids = spark.sparkContext.
        broadcast(centroids)
```

2.4.1 Risultati con parallelizzazione

L'esecuzione parallela migliora notevolmente le perfomance dell'algoritmo. La figura 7 mostra il risultato del clustering con KMEANS parallelo estraendo 15 campioni per ogni dataset. In entrambe le versioni (parallela e non parallela),

è fondamentale l'inizializzazione dei centroidi. Se i centroidi iniziali sono troppo vicini tra loro, l'algoritmo può convergere rapidamente su un solo cluster o distribuirli in modo non uniforme. Nella versione parallela, le iterazioni si comportano in modo più robusto data la distribuzione dei dati su più processi paralleli, consentendo più esplorazione nell'aggiornamento dei centroidi. Per questo motivo, i cluster risultanti sono più accurati e bilanciati.

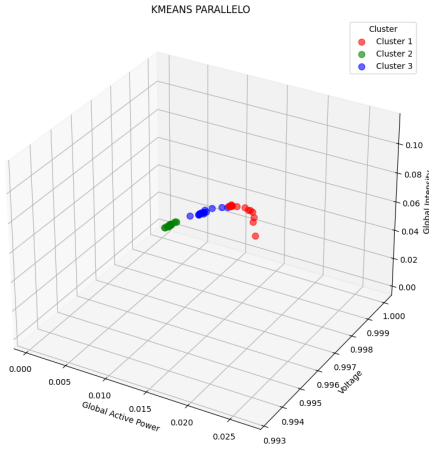


Figura 7: Risultati KMEANS parallelo

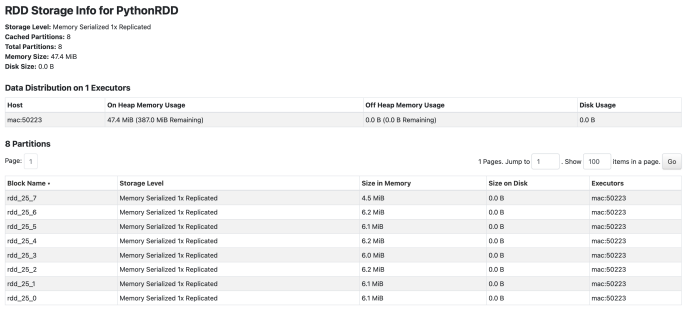
Il tempo di esecuzione medio risulta essere di 50 secondi circa. La riduzione del tempo di esecuzione rispetto all'implementazione seriale è un altro grande vantaggio della parallelizzazione. Per ottenere questi risultati, l'implementazione ha fatto uso della divisione del lavoro su più partizioni, come indicato nel seguente frammento di codice:

```
num_partitions = normalized_rdd.getNumPartitions()
```

Il numero di partizioni utilizzate è 8. Questo numero dipende dai meccanismi di ottimizzazione che applica Spark, dalla disponibilità della macchina su cui viene eseguito il codice e le prestazioni della rete. In un contesto distribuito o in sistemi con più core, Spark suddivide automaticamente il dataset in partizioni, cercando di bilanciare il carico tra i core per eseguire le operazioni in parallelo. La figura 8 mostra in dettaglio la suddivisione in partizioni creata da Spark, visualizzabile nell'interfaccia Spark UI. La pagina *Storage* mostra informazioni riguardo la memoria come la dimensione del dataset generato, il set di dati se memorizzato nella cache o meno e l'occupazione di RAM.

2.4.2 Analisi delle risorse

L'interfaccia *Spark UI* citata in precedenza e messa a disposizione da Spark permette di fare analisi sulle risorse occupate durante l'esecuzione del programma e di estrarre informazioni rilevanti. In primo luogo è possibile osservare come



RDD Storage Info for PythonRDD

Storage Level: Memory Serialized 1x Replicated
Cached Partitions: 8
Total Partitions: 8
Memory Size: 47.4 MiB
Disk Size: 0.0 B

Data Distribution on 1 Executors

Host	On Heap Memory Usage	Off Heap Memory Usage	Disk Usage
mac:50223	47.4 MiB (387.0 MiB Remaining)	0.0 B (0.0 B Remaining)	0.0 B

8 Partitions

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Block Name	Storage Level	Size in Memory	Size on Disk	Executors
rdd_25_7	Memory Serialized 1x Replicated	4.5 MiB	0.0 B	mac:50223
rdd_25_6	Memory Serialized 1x Replicated	6.2 MiB	0.0 B	mac:50223
rdd_25_5	Memory Serialized 1x Replicated	6.1 MiB	0.0 B	mac:50223
rdd_25_4	Memory Serialized 1x Replicated	6.2 MiB	0.0 B	mac:50223
rdd_25_3	Memory Serialized 1x Replicated	6.0 MiB	0.0 B	mac:50223
rdd_25_2	Memory Serialized 1x Replicated	6.2 MiB	0.0 B	mac:50223
rdd_25_1	Memory Serialized 1x Replicated	6.1 MiB	0.0 B	mac:50223
rdd_25_0	Memory Serialized 1x Replicated	6.1 MiB	0.0 B	mac:50223

Figura 8: Dataset salvato in cache KMEANS parallelo

Spark ha suddiviso il carico computazione sui jobs. Nel dettaglio sono stati eseguiti 15 jobs. Un job viene creato ogni volta che viene eseguita un'azione (*action*) su un RDD. L'immagine 9 mostra la timeline dell'action che sono state completate.

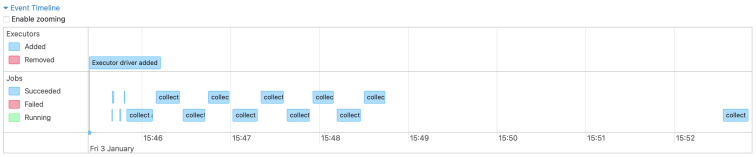


Figura 9: Timeline spark con Kmeans

Durante l'esecuzione, come mostrato in figura 10, si osserva una tabella con l'insieme degli stages con relative informazioni temporali e di memoria ma è possibile anche entrare nel dettaglio di ogni singola operazione. In particolare le colonne *shuffle read* e *shuffle write* mostrano la dimensione dei dati di shuffle sulla rete.

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
10	collect at /var/folders/l0/tnkwmzqx4b19vd3hsz_nsymw0000gn/T/ipykernel_3841/3986963273.py:31 +details	2025/01/03 16:04:08	59 ms	8/8			5.5 KiB	
9	reduceByKey at /var/folders/l0/tnkwmzqx4b19vd3hsz_nsymw0000gn/T/ipykernel_3841/3986963273.py:31 +details	2025/01/03 16:03:50	18 s	8/8	126.9 MiB			5.5 KiB
8	collect at /var/folders/l0/tnkwmzqx4b19vd3hsz_nsymw0000gn/T/ipykernel_3841/3986963273.py:31 +details	2025/01/03 16:03:50	73 ms	8/8			5.5 KiB	
7	reduceByKey at /var/folders/l0/tnkwmzqx4b19vd3hsz_nsymw0000gn/T/ipykernel_3841/3986963273.py:31 +details	2025/01/03 16:03:34	16 s	8/8	126.9 MiB			5.5 KiB
6	runJob at PythonRDD.scala:181 +details	2025/01/03 16:03:33	0,6 s	1/1	6.1 MiB			
5	runJob at PythonRDD.scala:181 +details	2025/01/03 16:03:32	0,2 s	1/1	4.1 MiB			
4	csv at NativeMethodAccessorImpl.java:0 +details	2025/01/03 16:03:30	0,5 s	8/8	127.4 MiB			
3	csv at NativeMethodAccessorImpl.java:0 +details	2025/01/03 16:03:30	4 ms	1/1	64.0 KiB			
2	runJob at PythonRDD.scala:181 +details	2025/01/03 16:03:15	1 s	1/1	4.1 MiB			
1	csv at NativeMethodAccessorImpl.java:0 +details	2025/01/03 16:03:12	1,0 s	8/8	127.4 MiB			
0	csv at NativeMethodAccessorImpl.java:0 +details	2025/01/03 16:03:12	23 ms	1/1	64.0 KiB			

Figura 10: Stages Spark UI del Kmeans

La scelta implementativa di caricare i dati in cache è gestita dal programmatore e permette di migliorare ulteriormente le performance di esecuzione del codice. Infatti l'algoritmo eseguito senza il passaggio del dataset in cache rima-

ne comunque molto efficiente (circa 2 minuti e 20 in media) ma meno rispetto a quest'ultima implementazione. Il codice che permette di caricare in cache il dataset è il seguente:

```
normalized_rdd.cache()
```

Dove *normalized_rdd* rappresenta la variabile dei dati normalizzati memorizzati in RDD. Infine, nella pagine Spark UI è possibile osservare informazioni sull'ambiente in cui viene eseguito il programma, sugli esecutori (*Executors*) e dettagli sulle query SQL eseguite.

3 DBSCAN

3.1 Introduzione dell'algoritmo

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) è un algoritmo di clustering che appartiene alla famiglia dei density-based model. Si basa sulla densità ed è progettato per identificare gruppi di punti in un insieme di dati che sono vicini tra loro nello spazio, oltre a identificare eventuali outlier (rumore). A differenza di algoritmi come K-means visto nella sezione precedente, DBSCAN non richiede di specificare a priori il numero di cluster, ma si basa sulla densità dei dati per individuarli. L'idea generale quindi è quella di andare a cercare, per ciascun punto, un'area circostante che contiene un numero minimo di punti ($MinPts$), dato uno specifico raggio (come *epsilon*). I punti con densità superiore ad un certo numero di $MinPts$ vengono chiamati punti core. Se la densità, attorno ad un punto, non supera il numero $MinPts$ richiesto e se un punto core si trova a una distanza minore di *epsilon*, allora questi punti vengono classificati come punti limite. I punti che non entrano in queste classificazioni vengono considerati come rumore. La figura 11 mostra l'idea generale dell'algoritmo, dove infatti i punti rappresentati con il colore nero sono considerati rumore.

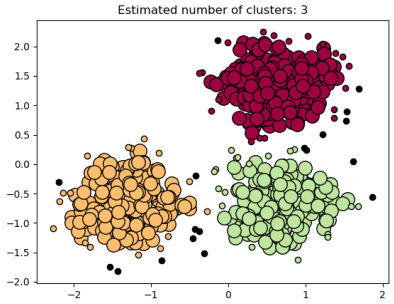


Figura 11: DBSCAN idea generale

3.2 Scelte implementative

Per algoritmo DBSCAN si è scelto di sviluppare implementazione standard e la sua versione parallela. L'algoritmo DBSCAN non dipende dal numero di features nei dati, ma piuttosto si concentra sulla distanza tra i punti nel loro spazio multidimensionale. Per quanto riguarda la scelta del valore *eps* si rimanda alle singole implementazioni.

3.3 Implementazione standard

L'approccio standard permette di visualizzare un caso specifico per il quale la parallelizzazione diventa obbligatoria. Si esegue l'algoritmo sull'1% del dataset per verifica il corretto funzionamento del codice:

```
dbscan = DBSCAN(eps=0.3, min_samples=10)
clusters = dbscan.fit_predict(features_scaled)
```

Si ottiene una classificazione con i seguenti risultati che dimostra che il codice è corretto:

- Outliers (-1): 168 punti
- Cluster 0: 20307 punti
- Cluster 1: 11 punti

Ma se aumenta il numero di dati l'algoritmo non termina e porta il kernel in crash come mostrano nell'immagine 12. Questo risultato si ottiene con diversi i valori di *epsilon* e $MinPts$ testati e utilizzando anche solo il 10% del dataset. Tale problema è generato dalla complessità dell'algoritmo e dalla dimensione del dataset, eccessiva per essere gestita in modo sequenziale. Pertanto, l'adozione di un approccio parallelo risulta necessaria per il corretto funzionamento dell'algoritmo.

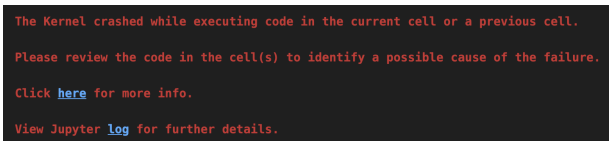


Figura 12: Immagine kernel in crash

3.4 Implementazione parallela

L'idea alla base della parallelizzazione di questo algortimo è dividere l'insieme dei dati in RDD per parallelizzare il calcolo e combinare i risultati ottenuti dai vari batch per generare il clustering finale. Per ogni RDD l'algoritmo prende ogni punto P e se questo non è stato *visitato* trova tutti i punti vicini a P entro

una distanza *epsilon*. Questo valore determina la distanza massima tra due punti affinché possano essere considerati vicini. Successivamente l'algoritmo valuta se il punto in esame ha un numero di vicini inferiore alla soglia $MinPts$, in tal caso segna il punto come rumore (outlier), altrimenti crea un nuovo cluster e aggiunge P e tutti i punti raggiungibili da P ad esso. Il codice sviluppato per la parallelizzazione implementa i seguenti step:

- Caricare i dati, configurare i parametri e inizializzare le variabili
- Allocare i punti dati nelle partizioni tramite metodi appropriati
- Eseguire DBSCAN locale in ogni partizione
- Unire i risultati da ogni partizione e restituirli come risultato finale

La funzione *process_point* viene applicata a ogni punto e ogni partizione esegue essenzialmente una versione locale di DBSCAN focalizzata sui punti di quella partizione. La funzione *ExpandCluster* implementa l'espansione del cluster, seguendo il metodo DBSCAN standard. Le variabili *epsilon* e *minPoints* sono passate come parametri e quindi condivise tra i worker. I risultati dei cluster locali vengono raccolti con *collect()* e uniti in un'unica lista di cluster come risultato finale seguendo il flusso descritto nell'immagine 13.

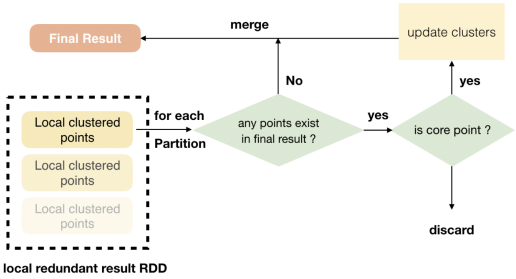


Figura 13: Fase di merge DBSCAN parallelo

Di seguito si mostra il codice che permette lo sviluppo dell'algortimo DBSCAN parallelo con Spark:

```
def process_point(point):
    if tuple(point) not in visited:
        visited.add(tuple(point))

        cluster = ExpandCluster(point,
                                epsilon,
                                minPoints,
                                data,
                                visited)
```

```
            if cluster:
                return cluster
            else:
                noise.add(tuple(point))
    return None
```

Le variabili *epsilon* e *minPoints* assegnate per ottenere i risultati mostrati in seguito sono state fissate rispettivamente a: $epsilon = 3.5$ e $minPoints = 10$. La seguente combinazione consente all'algoritmo di trovare cluster significativi senza fare errori, rispettando la distribuzione e la densità dei dati nel caso in esame. Si è scelto di applicare una PCA ai dati con ausilio di Spark per ridurre la dimensionalità dei dati a 2 componenti principali e migliorare notevolmente le performance, in particolare il tempo di esecuzione.

3.4.1 Risultati con parallelizzazione

Queste scelte implementative, hanno permesso di abbassare il tempo a circa 20 secondi. In figura 14 si mostra il risultato dell'algoritmo su una percentuale del 10% dei dati. Si osserva che il numero di cluster generati è pari a 49.

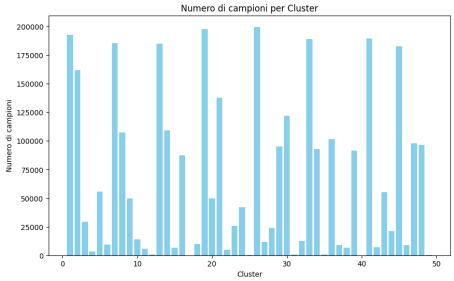


Figura 14: Grafico DBSCAN parallelo

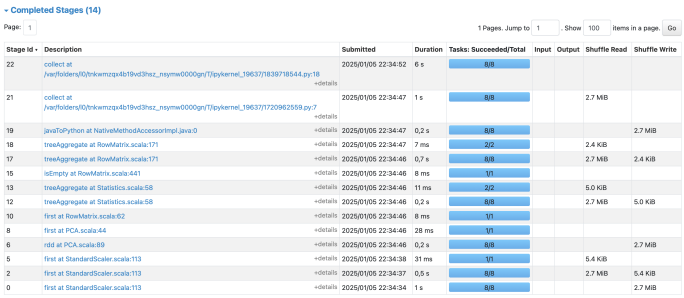
L'algoritmo DBSCAN rimane comunque complesso da risolvere, e si segnala che esecuzione completa sul dataset completo richiede molto tempo per arrivare al termine. Si conclude dunque che la parallelizzazione ha permesso di eseguire l'algoritmo e di ottenere un risultato che con il processo sequenziale non era ottenibile, inoltre Spark permette di distribuire il lavoro per aumentare il numero di dati per i quali è possibile arrivare all'esecuzione corretta.

3.4.2 Analisi delle risorse

Come descritto in precedenza, la UI di Spark permette di raccogliere informazioni. Le partizioni generate risultano essere 8. L'esecuzione invece porta a termine:

- Completati 12 jobs
- Completati 14 stages

Risultato coerente perché uno stage rappresenta una fase intermedia di un job. Un job può essere suddiviso in più stages se ci sono operazioni che richiedono una shuffle dei dati. Si mostra in figura 15 l'insieme degli stages dell'algoritmo DBSCAN parallelo con i valori di memoria nelle colonne *shuffle read* e *shuffle write*.



▾ Completed Stages (14)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
22	collect at /var/folders/l0/tnkwrmzqx4b19vd3hsz_nsymw0000gn/T/ipykernel_19637/1839718544.py:18 +details	2025/01/05 22:34:52	6 s	8/8				
21	collect at /var/folders/l0/tnkwrmzqx4b19vd3hsz_nsymw0000gn/T/ipykernel_19637/1720962559.py:7 +details	2025/01/05 22:34:47	1 s	8/8			2.7 MiB	
19	javaToPython at NativeMethodAccessorImpl.java:0 +details	2025/01/05 22:34:47	0,2 s	8/8				2.7 MiB
18	treeAggregate at RowMatrix.scala:171 +details	2025/01/05 22:34:47	7 ms	2/2			2.4 KiB	
17	treeAggregate at RowMatrix.scala:171 +details	2025/01/05 22:34:46	0,7 s	8/8			2.7 MiB	2.4 KiB
15	isEmpty at RowMatrix.scala:441 +details	2025/01/05 22:34:46	8 ms	1/1				
13	treeAggregate at Statistics.scala:58 +details	2025/01/05 22:34:46	11 ms	2/2			5.0 KiB	
12	treeAggregate at Statistics.scala:58 +details	2025/01/05 22:34:46	0,2 s	8/8			2.7 MiB	5.0 KiB
10	first at RowMatrix.scala:62 +details	2025/01/05 22:34:46	8 ms	1/1				
8	first at PCA.scala:44 +details	2025/01/05 22:34:46	28 ms	1/1				
6	rdd at PCA.scala:89 +details	2025/01/05 22:34:46	0,2 s	8/8				2.7 MiB
5	first at StandardScaler.scala:113 +details	2025/01/05 22:34:38	31 ms	1/1			5.4 KiB	
2	first at StandardScaler.scala:113 +details	2025/01/05 22:34:37	0,5 s	8/8			2.7 MiB	5.4 KiB
0	first at StandardScaler.scala:113 +details	2025/01/05 22:34:34	1 s	8/8				2.7 MiB

Figura 15: Stages Spark UI DBSCAN parallelo

4 Gaussian Mixture Models (GMM)

4.1 Introduzione all'algoritmo

L'algoritmo Gaussian Mixture Models (GMM) è una tecnica di soft clusteting molto utilizzata. Appartiene alla classe dei metodi non supervisionati e permette di identificare cluster o gruppi di dati assumendo che questi possono essere descritti da una distribuzione normale con specifica media e varianza. L'algoritmo, dunque, si mostra ottimo da applicare al dataset del progetto per scovare gruppi di dati che mostrano consumi con distribuzioni simili. GMM usa l'*algoritmo EM* per ottimizzare le sue performance. In particolare questo algoritmo itera tra due fasi principali:

- *Expectation (E-step)*: calcola la probabilità (responsabilità) che ogni punto appartenga a ciascun cluster dato i parametri correnti e utilizzando la densità di probabilità pesata di ogni punto.
- *Maximization (M-step)*: aggiorna i parametri del modello (le medie, le covarianze, e le probabilità dei cluster) in base alle responsabilità calcolate nella E-step.

Si mostrano gli step utilizzati per sviluppare l'algoritmo:

- Inizializzazione
  - Le medie sono inizializzate selezionando campioni casuali dal dataset.
  - Le covarianze sono inizializzate come matrici identità.
  - I pesi iniziali sono uguali per ogni cluster.

- *E-step*
- *M-step*
- *Log-Likelihood*: calcola il log-likelihood totale per monitorare la convergenza dell'algoritmo. Il ciclo termina quando il miglioramento della log-likelihood scende sotto una soglia di tolleranza (*tol*) oppure raggiunge il numero massimo di iterazioni (max_iter).

Di seguito riportato il codice che implementa l'algoritmo richiamando le funzioni appena descritte:

```
for iteration in range(n_iter):

    # E - STEP
    responsibilities = e_step(data,
                              means,
                              covariances,
                              weights)

     # M - STEP
    means, covariances, weights =
        m_step(data, responsibilities)

    # LIKELIHOOD
    log_likelihood = compute_log_likelihood(data,
                                        means,
                                        covariances,
                                        weights)

    # CONTROLLO DELLA CONVERGENZA
    if np.abs(log_likelihood -
                prev_log_likelihood) < tol:
        break

    prev_log_likelihood = log_likelihood
```

4.2 Scelte implementative

GMM è molto sensibile alla scelta del numero di componenti e numero di iterazioni. Nello stato dell'arte sono presenti diverse tecniche per calcolare quali valori ottengono performance e risultati più accurati per l'algoritmo. Fra le più diffuse si trovano: Akaike information criterion, Bayesian information criterion, Cross-Validation. Dopo aver testato diversi parametri e per semplicità di implementazione si è scelto di inizializzare le variabili con valori standard come segue:

```
n_components = 3
n_iter = 100
tol = 1e-4
```

In tutte le versioni implementate dell'algoritmo si è scelto di usare il metodo *K-means++* per ottenere una distribuzione migliore dei centroidi iniziali.

4.2.1 K-means++

Nel K-means classico, i centroidi iniziali vengono scelti casualmente da un insieme di punti nel dataset. Questo approccio può portare a una selezione di centroidi che non rappresentano bene la distribuzione dei dati, influenzando negativamente la convergenza e la qualità del clustering. Nel K-means++, invece, i centroidi iniziali sono scelti in modo più strategico:

- Si sceglie un primo centroide casualmente
- Per ciascun punto rimanente, si calcola la distanza al centroide più vicino e si assegna a ogni punto una probabilità di essere scelto come nuovo centroide in funzione di questa distanza (più lontano dal centroide attuale, più alta la probabilità di essere scelto)
- Si ripete il passo 2 fino a che sono stati scelti tutti i centroidi

Questo approccio aiuta a garantire che i centroidi iniziali siano distribuiti in modo più uniforme e che rappresentino meglio la struttura dei dati e quindi migliorare la qualità del clustering iniziale e, di conseguenza, la convergenza dell'algoritmo.

4.3 Implementazione standard

L'algoritmo GMM nella sua versione standard descritto in precedenza viene dunque applicato al dataset in esame. Si osserva che il tempo di esecuzione è di circa 1 minuto e la convergenza si ottiene raggiungendo in genere le 100 iterazioni. Nelle figure 16 e 17 si osserva l'output dell'esecuzione.

I risultati mostrati si basano sull'esecuzione dell'algoritmo con uno sviluppo manuale delle singole funzioni che implementano i singoli passi. Utilizzano la funzione appartenente alla libreria *sklearn* il tempo di esecuzione risulta essere in media 40 secondi. Si osserva, invece, che i risultati sono coerenti con quelli precedenti, i cluster risultano essere posizionati negli stessi punti e la distribuzione è la stessa. Le figure 19 e 18 mostrano i risultati graficamente.

È importante sottolineare che l'esecuzione con supporto della libreria in automatico esegue ottimizzazioni delle performance e per questo migliora il tempo di esecuzione. Lo sviluppo parallelo con Spark è applicabile per sfruttare l'uso di partizioni e la memoria distribuita. Si consente dunque al modello di lavorare più velocemente su set di dati complessi, migliorando le prestazioni e in particolare l'accuratezza. Inoltre il *metodo EM* è particolarmente adatto per essere applicato all'architettura distribuita di Spark.

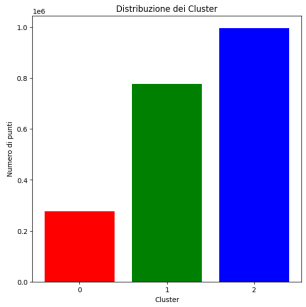


Figura 16: Distribuzione dei cluster GMM standard

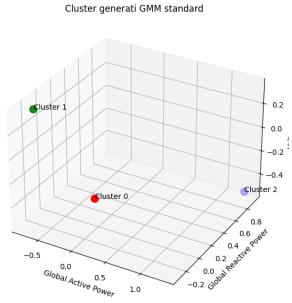


Figura 17: Rappresentazione grafica dei cluster

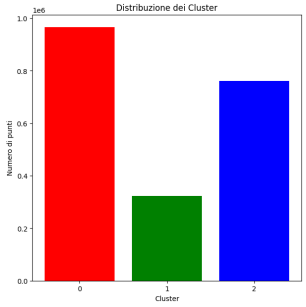


Figura 18: Distribuzione dei cluster GMM da libreria

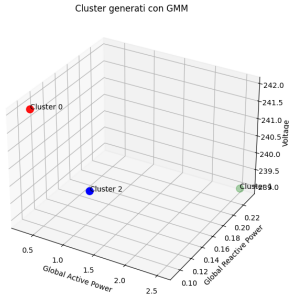


Figura 19: Rappresentazione grafica dei cluster

4.4 Implementazione parallela

Lo sviluppo parallelo dell'algoritmo GMM sfrutta la libreria di machine learning di Spark, nel dettaglio il codice è:

```
pca = PCA(k=2, inputCol="features",
    outputCol="pca_features")
pca_model = pca.fit(df_spark)
df_spark = pca_model.transform(df_spark)

gmm = GaussianMixture(k=3, maxIter=100,
    tol=1e-4, seed=42)
model = gmm.fit(df_spark)
```

Si applica la PCA per ridurre la dimensionalità dei dati, proiettando i dati su un nuovo spazio di dimensioni inferiori, in questo caso a 2 dimensioni, per ottenere miglioramento delle prestazioni, distanza e separabilità dei dati più accurata e stabilità dell'algoritmo. Infine, sempre in termini di performance, il dataset viene caricato in cache e salvato nel formato Spark DataFrame.

4.4.1 Risultati con parallelizzazione

La parallelizzazione ha permesso in primo luogo di diminuire il tempo di esecuzione, portandola all'incirca a 22 secondi. Tempo dimezzato rispetto all'esecuzione non parallela. Come si osserva nelle immagini 20 e 21 la distribuzione dei dati risulta più accurata.

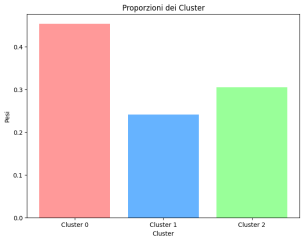


Figura 20: Distribuzione dei cluster GMM da libreria

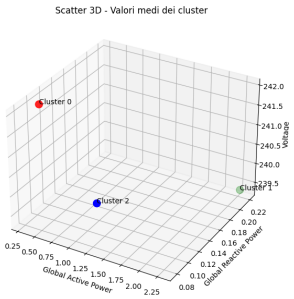


Figura 21: Rappresentazione grafica dei cluster

Si mostra nell'immagine 22 la distribuzione della variabile *Voltage* nel dataset, suddivisa per i tre cluster identificati dal modello di GMM.

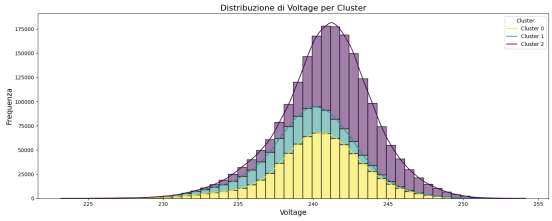


Figura 22: Distribuzione della variabile *Voltage*

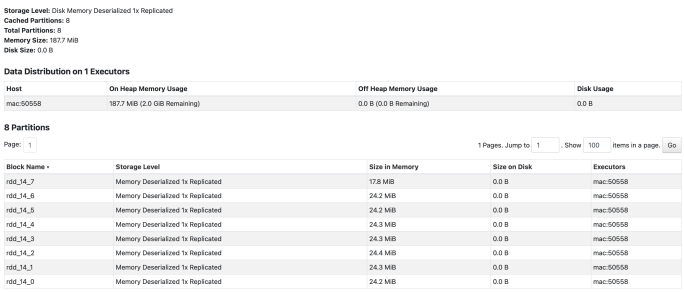
4.4.2 Analisi delle risorse

Si visualizzano attraverso SPARK UI le risorse utilizzate per l'esecuzione dell'algoritmo GMM parallelo. Spark porta a termine 214 stages per 112 jobs finali. L'immagine 23 mostra gli esecutori:.

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Excluded
Active(1)	8	187.8 MiB / 2.2 GiB	0.0 B	8	0	0	1665	1665	3.1 min (0.5 s)	21.9 GiB	976.9 KiB	976.9 KiB	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	0
Total(1)	8	187.8 MiB / 2.2 GiB	0.0 B	8	0	0	1665	1665	3.1 min (0.5 s)	21.9 GiB	976.9 KiB	976.9 KiB	0

Figura 23: Esecutori algoritmo GMM parallelo

Le partizioni utilizzate risultano essere 8. Si mostrano dunque i valori di memoria occupata con la sezione storage rappresentata in figura 24.



Storage Level: Disk Memory Deserialized 1x Replicated
Cached Partitions: 8
Total Partitions: 8
Memory Size: 187.7 MiB
Disk Size: 0.0 B

Data Distribution on 1 Executors

Host	On Heap Memory Usage	Off Heap Memory Usage	Disk Usage
mac:50558	187.7 MiB (2.0 GiB Remaining)	0.0 B (0.0 B Remaining)	0.0 B

8 Partitions

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Block Name	Storage Level	Size in Memory	Size on Disk	Executors
rdd_14_7	Memory Deserialized 1x Replicated	17.8 MiB	0.0 B	mac:50558
rdd_14_6	Memory Deserialized 1x Replicated	24.2 MiB	0.0 B	mac:50558
rdd_14_5	Memory Deserialized 1x Replicated	24.2 MiB	0.0 B	mac:50558
rdd_14_4	Memory Deserialized 1x Replicated	24.3 MiB	0.0 B	mac:50558
rdd_14_3	Memory Deserialized 1x Replicated	24.3 MiB	0.0 B	mac:50558
rdd_14_2	Memory Deserialized 1x Replicated	24.4 MiB	0.0 B	mac:50558
rdd_14_1	Memory Deserialized 1x Replicated	24.3 MiB	0.0 B	mac:50558
rdd_14_0	Memory Deserialized 1x Replicated	24.2 MiB	0.0 B	mac:50558

Figura 24: Storage algoritmo GMM

5 Conclusioni

Gli algoritmi di clustering permettono di raccogliere insieme di dati che esprimono informazioni simili in classi omogenee. Nel contesto del dataset in oggetto è possibile definire classi di consumi energetici, definire classi di oggetti per livello di consumo, visualizzare anomalie e consumi anomali. Lavorare su insieme molto ampio di dati in maniera sequenziale si è dimostrato essere un processo lento, poco affidabile e difficile da supportare per casi reali. Dunque, la parallelizzazione si rende estremamente utile per migliorare le performance, l'accuratezza e i risultati degli algoritmi analizzati. In generale i vantaggi di una parallelizzazione su grandi quantità di dati sono positivi in molte situazioni, le quali rendono strumenti come Spark la base della corretta esecuzione di processi reali.

6 Bibliografia

- Figura 13 `https://cse.hkust.edu.hk/msbd5003/pastproj/deep1.pdf`
- Figura 11 `https://scikit-learn.org/dev/modules/clustering.html`
- `https://it.wikipedia.org/wiki/Clustering`
- `https://cloud.google.com/learn/what-is-apache-spark?hl=it`
- `https://spark.apache.org`
- Articolo `https://cse.hkust.edu.hk/msbd5003/pastproj/deep1.pdf`
- Dataset `https://archive.ics.uci.edu/dataset/235/individual+household+electric+power+consumption`
- Libreria scikit-learn `https://scikit-learn.org/stable/`

- Slide corso "Big Data", UniVR