Sistemi e Architetture per Big Data - AA 2019/2020

Progetto 1: Analisi del dataset Covid-19 con Spark

Marco Balletti
Dipartimento di Ingegneria dell'Informazione
Università degli studi di Roma "Tor Vergata"
Roma, Italia
marco.balletti@alumni.uniroma2.eu

Francesco Marino
Dipartimento di Ingegneria dell'Informazione
Università degli studi di Roma "Tor Vergata"
Roma, Italia
francesco.marino.412@alumni.uniroma2.eu

ABSTRACT

Questo documento riporta i dettagli implementativi riguardanti l'analisi mediante Spark del *dataset* contenente informazioni riguardanti l'andamento nazionale italiano di casi positivi e tamponi e del *dataset* riguardante l'andamento globale dei contagi per ogni nazione. Verranno descritte anche l'architettura a supporto dell'analisi e gli ulteriori *framework* utilizzati.

KEYWORDS

• Covid • Contagi • Tamponi • Statistiche • Clustering • K-Means • Spark • MLlib • Lloyd • HBase • HDFS • NiFi • Docker

1 Introduzione

L'analisi effettuata si pone lo scopo di valutare delle statistiche relative alla pandemia COVID-19 sia su dati globali che italiani.

Il primo *dataset* preso in considerazione è `dpc-covid19-ita-andamento-nazionale` (disponibile all'URL [1]) che contiene dati giornalieri riguardanti l'andamento del COVID-19 in Italia dal 24 Febbraio 2020. Il *dataset*, in particolare, si compone dei seguenti campi: `data` (nel formato yyyy-MM-ddThh:mm:ss), `stato` (questo campo è fisso ad ITA in tutto il dataset), `ricoveri con sintomi` (dato cumulativo), `terapia intensiva` (dato cumulativo), `totale ospedalizzati` (dato cumulativo), `isolamento domiciliare` (dato cumulativo), `totale positivi` (dato cumulativo), `variazione totale positivi`, `nuovi positivi`, `dimessi guariti` (dato cumulativo), `deceduti` (dato cumulativo), `totale casi` (dato cumulativo), `tamponi` (dato cumulativo), `casi testati` (dato cumulativo), `note it` e `note en`.

Il secondo *dataset* preso in considerazione è `time_series_covid19_confirmed_global` (disponibile all'URL [2]) che raccoglie i dati cumulativi dei casi confermati in ogni stato dal 22 Gennaio 2020. Ogni riga del *dataset* è composta da `province/state` (potrebbe essere vuoto), `country/region`, `lat` (latitudine), `lon` (longitudine) e una serie di colonne, una per ogni giorno, con i dati (cumulativi) registrati.

L'analisi, in particolare, si compone delle tre *query* elencate di seguito.

1.1 Query 1

Per ogni settimana, si chiede di calcolare il numero medio di guariti e dei tamponi effettuati in Italia in quella settimana.

1.2 Query 2

Per ogni continente, si chiede di calcolare la media, la deviazione standard, il minimo e il massimo del numero di casi confermati giornalmente per ogni settimana. Nel calcolo delle statistiche, si richiede di considerare solo i 100 stati più colpiti dalla pandemia. Qualora lo stato non fosse indicato, risulta necessario considerare la nazione. Per determinare gli stati più colpiti nell'intero *dataset*, si considera l'andamento degli incrementi giornalieri dei casi confermati attraverso il *trendline coefficient*. Per stimare tale coefficiente, è necessario calcolare la pendenza della retta di regressione che approssima la tendenza degli incrementi giornalieri.

1.3 Query 3

Considerando i 50 stati più colpiti calcolati su base mensile secondo il *trendline coefficient*, per ogni mese nel *dataset* si richiede di applicare l'algoritmo di *clustering* K-means con K = 4 (numero di *cluster*) determinando gli stati (o nazioni) che fanno parte di ogni *cluster*. Ogni *cluster*, quindi, raggrupperà gli stati che hanno un simile andamento degli incrementi giornalieri dei casi confermati nel mese preso in considerazione.

2 Architettura

L'architettura utilizzata per l'analisi dei dati si compone di una JVM (*Java Virtual Machine*) locale su cui viene eseguito il *framework* di *batch processing* Spark e di diversi *container* Docker su cui sono stati installati il *distributed*

filesystem HDFS, il *datastore NoSQL* Hbase, il *framework* di *data ingestion* NiFi, il *database* orientato a serie temporali InfluxDB ed il *framework* Grafana per la visualizzazione dei risultati. Tutti i *container* Docker comunicano tra loro mediante una rete interna creata appositamente, per permettere al *framework* Spark di raggiungere i dati presenti nell'HDFS è necessario che il sistema operativo *host* permetta di effettuare il *routing* verso indirizzi IP appartenenti alla sottorete di Docker. Questa condizione è verificata se si utilizza Linux ma non se il sistema operativo *host* è macOS (a causa di dettagli implementativi relativi all'applicazione Docker Desktop).

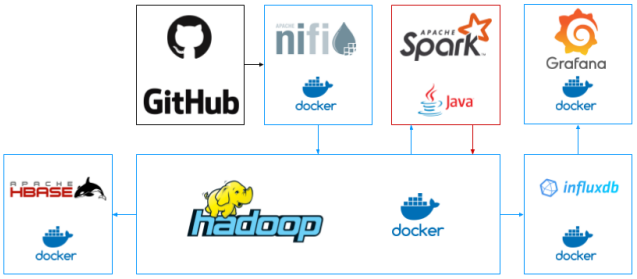

Figura 1: Schema dell'architettura

Di seguito verranno descritte le configurazioni e l'architettura di ogni componente.

2.1 NiFi

NiFi, istanziato su un *container* Docker creato a partire dall'immagine `apache/nifi`, è stato utilizzato per eseguire il *pre-processing* e l'*import* nell'HDFS dei *dataset* presentati precedentemente.

Le operazioni eseguite sul primo *dataset*, in ordine, sono:

- *download* del primo *dataset* dal *repository* GitHub,
- *splitting* del *dataset* in righe per eseguire il *pre-processing* su ogni riga separatamente,
- eliminazione, da ogni riga, delle colonne non necessarie (in particolare sono state mantenute solamente le colonne `data`, `totale_positivi` e `tamponi`),
- *merging* delle righe in un unico *file* (mantenendo lo stesso ordine che si aveva prima dello *splitting*),
- rimozione della prima riga del file contenente l'*header*,
- rimozione delle linee vuote (alcuni aggiornamenti del *dataset* hanno introdotto delle righe vuote alla fine del file che creavano problemi al processamento) e
- *import* del *dataset* pre-processato nell'HDFS.

Le operazioni eseguite, invece, sul secondo *dataset*, in ordine, sono:

- *download* del secondo *dataset* dal *repository* GitHub,
- rimozione della prima riga del *file* contenente l'*header*,
- rimozione di eventuali linee vuote,
- rimozione delle virgole dai nomi di nazioni e province (lo *split* di ogni riga durante l'analisi, in presenza di virgole nei nomi, generava, infatti, un numero sbagliato di elementi) e
- *import* del *dataset* pre-processato nell'HDFS.

Di seguito vengono riportati graficamente i *flow* di NiFi descritti precedentemente per mostrare un accenno di interfaccia grafica offerta dal *framework*.

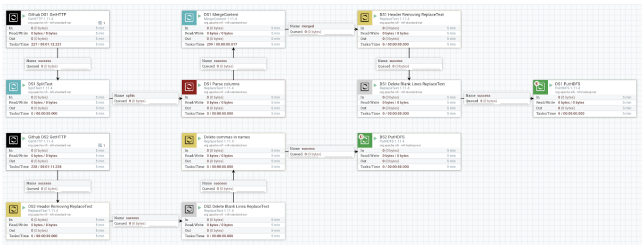

Figura 2: Flow realizzato in NiFi

Il flusso generato in NiFi è stato reso persistente eseguendo un *binding* delle *directory* contenenti lo stato e la configurazione di NiFi a dei volumi esterni. Per poter permettere anche un'importazione semplice del flusso in nuovi *container*, il NiFi *flow* è stato anche esportato in formato `.xml` e memorizzato in una apposita *directory* del progetto.

Per poter permettere a NiFi di comunicare correttamente con l'HDFS sono stati importati all'interno del *container* i *file* `core-site.xml` e `hdfs-site.xml` (esportati dal *container* contenente il *Namenode* del *filesystem* distribuito) ed i rispettivi *path* sono stati passati in argomento ai *flow file processor* che si occupano della scrittura sul *filesystem*.

2.2 HDFS

L'*Hadoop Distributed Filesystem* è stato istanziato utilizzando dei *container* Docker, in particolare sono stati creati, a partire dall'immagine `effeerre/hadoop`, 4 *container*: il primo contenente il *Namenode* dell'HDFS ed i restanti 3 i *Datanode*. La funzione del *filesystem* distribuito è quella di memorizzare i dati processati da NiFi, che costituiranno l'*input* dei processamenti Spark, e gli *output* dell'analisi che poi verranno ripresi per essere esportati verso HBase. Per permettere la corretta esecuzione di

queste operazioni è stato realizzato uno *script* che si occupa di:

- formattare lo stato ed il contenuto del *filesystem* distribuito,
- eseguire lo *startup* del *Namenode* e dei *Datanode*,
- creare una *directory* (`/data`) dove i *file* pre-processati saranno inseriti da NiFi,
- concedere i permessi di lettura, scrittura ed esecuzione su tale *directory* per permettere a NiFi di importare i dati,
- creare una *directory* (`/output`) dove saranno salvati i risultati del processamento Spark e
- concedere i permessi di lettura, scrittura ed esecuzione su tale *directory* per permettere a Spark e Hbase di scriverci e leggerci i dati.

Tale *script* viene importato nel *container* su cui sarà eseguito il nodo *master* dell'HDFS ed avviato automaticamente subito dopo la creazione di tutti i rimanenti *container* necessari per il funzionamento del *filesystem*.

2.3 Spark

Spark viene eseguito su una singola JVM (*Java Virtual Machine*). Oltre allo Spark Core sono stati utilizzati anche Spark MLlib per effettuare *clustering* mediante l'algoritmo K-Means e Spark SQL per eseguire l'analisi richiesta nelle *query* 1 e 2 in modo differente rispetto alle classiche trasformazioni ed azioni dello Spark Core.

L'*import* di Spark e delle sue estensioni, tutte e tre nella versione 2.12, è stato eseguito utilizzando Maven.

2.4 HBase

Hbase è stato utilizzato come *datastore NoSQL* sul quale importare i risultati delle *query*, anche questo servizio è stato istanziato utilizzando un *container* Docker realizzato, questa volta, a partire dall'immagine `harisekhon/hbase`. Al fine di gestire la comunicazione con Hbase sono state create due classi Java.

La prima delle due classi, chiamata `HBaseLightClient`, contiene indirizzi e porte di Hbase e Zookeeper, le funzioni per la creazione e l'eliminazione delle tabelle di Hbase, per l'aggiunta di una nuova riga ad una tabella esistente e per la stampa del contenuto di una tabella.

La seconda classe, chiamata `HbaseImport`, consiste di un metodo `main` e di tre funzioni per il caricamento dei risultati delle varie query dall'HDFS ad Hbase. Nello specifico, il `main` si occupa di istanziare un `HBaseLightClient` tramite il quale creare le tabelle per poi richiamare le tre funzioni che si occupano di:

- aprire un collegamento in lettura verso i *file* con i risultati delle *query* (che si trovano nella *directory* dell'HDFS `/output`),
- eseguire il *parsing* dei *file* sfruttando le Java Regex,
- inserire nella tabella di Hbase relativa alla *query* correntemente analizzata i risultati ottenuti dal *parsing* e
- chiudere il collegamento precedentemente aperto.

2.5 InfluxDB

Il *datastore* orientato alle serie temporali, istanziato su un *container* Docker realizzato a partire dall'immagine `influxdb`, è stato utilizzato per trasferire i risultati delle *query* dall'HDFS verso il *framework* Grafana consentendone, in questo modo, la visualizzazione. Per eseguire questa operazione, è stato necessario convertire ogni risultato in serie temporale con dati intervallati mensilmente o settimanalmente (a seconda del tipo di risultato da importare).

Per garantire la persistenza del *database* nell'ambito della macchina *host* è stato eseguito il *binding* della *directory* contenente lo stato e la configurazione di InfluxDB; per garantire l'accesso al datastore sono anche stati impostati username e password.

Anche in questo caso, come si è fatto per Hbase, sono state create due classi Java per gestire la comunicazione con InfluxDB.

La prima classe, chiamata `InfluxDBClient`, contiene l'URL di InfluxDB, le credenziali di accesso (*username* e *password*) e le funzioni necessarie per la gestione e l'inserimento dei dati nel *datastore* orientato alle *time series*.

La seconda classe, chiamata `InfluxDBImport`, consiste, proprio come la sua analoga per Hbase, di un metodo `main` e di alcune funzioni per il caricamento dei risultati delle varie *query* dall'HDFS ad InfluxDB. Nello specifico, il metodo `main` si occupa di istanziare un `InfluxDBClient` tramite il quale creare il *database* per poi richiamare le funzioni per l'inserimento dei dati.

2.6 Grafana

Il *framework* di visualizzazione è stato istanziato su un *container* Docker basato sull'immagine `grafana/grafana`. Similmente ai casi precedenti, per implementare la persistenza del *container* nell'ambito della macchina *host* locale è stato eseguito il *binding* delle *directory* di Grafana ad un volume esterno. Sono state realizzate tre *dashboard*, una per ogni *query*, con diversi grafici all'interno; le stesse *dashboard* sono state salvate

come immagini e come file JSON (in due versioni: una per l'*import* in Grafana ed una per la condivisione su web) per semplificarne l'*import* in *container* su nuove macchine *host*. Rimane comunque necessario, come operazione preliminare all'*import* dei *file* JSON, impostare influxDB come sorgente di dati in Grafana. Di seguito un'immagine con le *dashboard* per mostrare l'interfaccia grafica messa a disposizione dal *framework*:

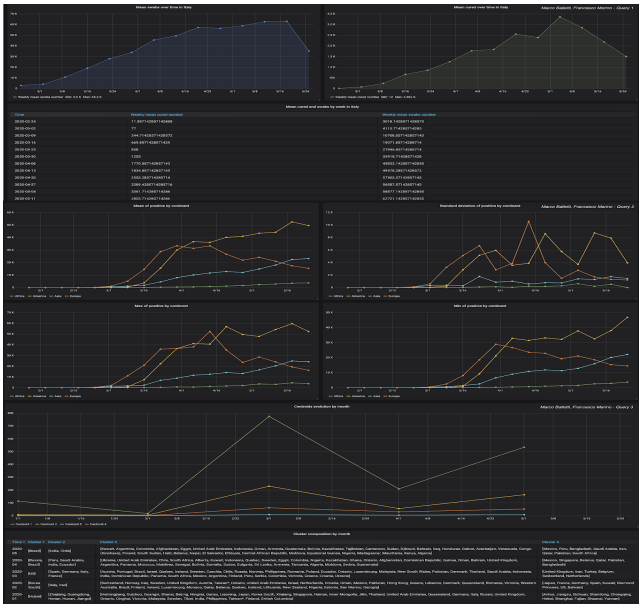

Figura 3: Dashboard realizzate in Grafana

3 Query 1 workflow

La prima *query* è stata strutturata in modo da poter essere eseguita sia usando solamente lo Spark Core sia sfruttando il *framework* di più alto livello Spark SQL; per fare questo viene eseguita una prima parte di pre-processamento dei dati comune ad ambe le soluzioni.

3.1 Pre-Processing

Questa parte di processamento si occupa di eseguire il *parsing* del CSV preso dall'HDFS al fine di ottenere un RDD che contenga le seguenti informazioni: data, tamponi effettuati (*swabs*) e persone guarite (*cured*).

Successivamente, questo RDD viene manipolato apportando le seguenti modifiche:

- la data viene trasformata nel primo giorno di quella settimana così da poter, in seguito, lavorare più facilmente su dati relativi alla stessa settimana,
- i valori di tamponi effettuati e persone guarite viene trasformato da dato cumulativo a dato giornaliero (qualora ci fossero errori nel *dataset* si è scelto di considerare il dato errato come mancante e, di conseguenza, come un incremento nullo rispetto al giorno precedente).

3.2 Soluzione Spark Core pura

Se si vuole sfruttare solamente lo Spark Core si prosegue calcolando la media di tamponi effettuati e persone guarite settimana per settimana raggruppando le righe dell'RDD rispetto alla settimana.

Una volta calcolate le statistiche, queste vengono scritte nella *directory* `/output/query1` dell'HDFS sfruttando il metodo `saveAsTextFile` degli RDD.

3.3 Soluzione Spark Core e Spark SQL

In questa soluzione, a differenza della precedente, si sfrutta anche il *framework* Spark SQL; si prosegue creando, a partire dai dati ottenuti dal pre-processamento, un *dataset* strutturato su quattro colonne: `id` (intero auto-incrementale), `week`, `cured` e `swabs`.

Su questo *dataset* si utilizzano le funzionalità built-in offerte da Spark SQL per il calcolo della media settimana per settimana.

4 Query 2 workflow

La seconda *query*, similmente alla prima, essendo strutturata per permettere l'esecuzione utilizzando solo lo Spark Core o anche Spark SQL, presenta una parte di pre-processamento (eseguito mediante le trasformazioni e le azioni proprie dello Spark Core) comune ad entrambi i flussi logici di analisi.

4.1 Pre-processing

In questa fase viene innanzitutto caricato il *dataset* dall'HDFS e trasformato in un RDD in cui ogni elemento corrisponde ad una riga del file originario. All'RDD sono quindi applicate le seguenti trasformazioni logiche sfruttando lo Spark Core:

- *parsing* di tutte le informazioni riguardanti l'elemento del *dataset* e trasformazione dei dati cumulativi in dati puntuali,
- calcolo del *trendline coefficient* (*slope*) a partire dai dati puntuali per ogni elemento del *dataset*,
- ordinamento degli elementi dell'RDD secondo il valore dello *slope* e riduzione del *set* ad i primi 100 elementi e
- computazione, a partire dalle coordinate geografiche, del continente.

La computazione del continente a partire da latitudine e longitudine sfrutta un duplice algoritmo le cui parti sono di seguito presentate.

4.1.1 Esecuzione locale [3]:

Per riconoscere il continente senza sfruttare risorse esterne al sistema locale si sfruttano delle *fence* poligonali rappresentative delle aree geografiche di ogni continente e si controlla se la coppia latitudine, longitudine si trova all'interno di ognuna. Per eseguire questa verifica in modo efficiente, si prende il punto rappresentato dalle due coordinate, da tale punto viene fatta partire, idealmente, una retta orizzontale che si estende verso + ∞, vengono contate, quindi, le volte che tale retta si interseca con il poligono rappresentante il continente. Se il numero di intersezioni è pari (o 0) il punto si trova all'esterno della *fence* (basti pensare che, essendo tale numero pari, sono tante le volte che si entra all'interno del poligono quante quelle che si esce dallo stesso), se è dispari è, invece, si trova all'interno. Una soluzione di questo tipo funziona abbastanza bene per il problema preso in esame nonostante alcune imprecisioni che potrebbero essere presenti nella definizione dei poligoni rappresentanti i continenti. Questo è vero poiché le latitudini e le longitudini presenti nel *dataset*, essendo rappresentative di stati o nazioni, non risultano essere troppo prossime ai confini dei continenti; allo stesso tempo, però, l'algoritmo può fallire nel caso di dati relativi ad isole perché, nel definire i poligoni, non sono stati inclusi gli oceani e le isole al loro interno.

4.1.2 Esecuzione tramite REST API:

Questo tipo di riconoscimento si basa sullo sfruttare i servizi di *reverse geocoding* messi a disposizione in rete. In particolare, viene inoltrata una richiesta HTTP verso uno di questi servizi specificando le coordinate geografiche e in risposta, tra le varie informazioni, si ottiene un codice identificativo del paese in cui tali coordinate ricadono. Sfruttando tali codici, è possibile, quindi, riconoscere il continente a cui le coordinate di partenza fanno riferimento. Durante lo sviluppo, sono state effettuate delle prove utilizzando due differenti servizi, entrambi gratuiti, di *reverse geocoding*, il primo offerto da Nominatim [4] ed il secondo da Big Data Cloud [5]. I test hanno mostrato che il secondo offre prestazioni di traduzione migliori in termini di *throughput* e latenza arrivando ad un tempo di risoluzione fino a 12 volte minore rispetto al primo (test effettuati su 260 richieste di *reverse geocoding*) ed è quindi stato scelto per essere integrato nella soluzione finale di traduzione dei continenti.

La soluzione finale adottata consiste, quindi, nel preferire una traduzione locale, poiché molto più efficiente in termini di latenza, e, laddove non fosse possibile determinare il continente a causa delle limitazioni precedentemente elencate, procedere con un'identificazione tramite servizi *web* di *reverse geocoding*. In termini di tempo di esecuzione, quest'ultima soluzione è risultata essere fino a 101 volte più veloce di quella che faceva utilizzo del solo servizio di *reverse geocoding* offerto da Big Data Cloud [5].

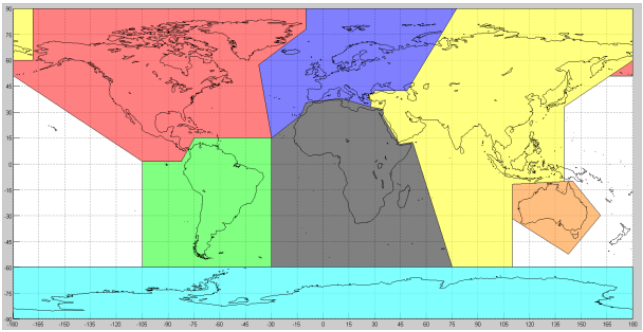

Figura 4: Definizione dei poligoni nella soluzione locale

4.2 Soluzione Spark Core pura

Nella soluzione pura, il flusso logico procede con le seguenti operazioni:

- suddivisione dei dati per settimana,
- aggregazione dei dati settimanali per gli stati o nazioni appartenenti allo stesso continente e
- calcolo delle statistiche richieste.

Una volta trovati i risultati, questi vengono salvati nella *directory* `/output/query2` dell'HDFS sfruttando il metodo `saveAsTextFile` degli RDD.

4.3 Soluzione Spark Core e Spark SQL

Il risultato del *pre-processing*, anche in questo caso, viene suddiviso per settimane e convertito in un *dataset* con cinque colonne: `id`, `continent`, `week`, `day` (numero di giorno nella settimana), `positive` (numero di casi confermati).

Su tale *dataset* vengono eseguite due *query* Spark SQL: la prima consiste nel sommare il numero di casi relativi allo stesso continente e allo stesso giorno della settimana, la seconda nel calcolare, infine, le statistiche richieste.

5 Query 3 workflow

La terza *query* consiste di un processamento dei dati e dell'applicazione, su questi, dell'algoritmo K-means.

Nella fase iniziale, si provvede al caricamento del *dataset* dall'HDFS ed alla conversione dello stesso in RDD. L'RDD

generato è soggetto, quindi, alle seguenti operazioni logiche:

- *parsing* di tutte le informazioni,
- trasformazione dei dati cumulativi in dati puntuali,
- suddivisione dei dati su base mensile,
- calcolo del *trendline coefficient* (*slope*) mensile per ogni stato,
- mantenimento dei soli 50 stati che, mese per mese, risultano avere lo *slope* più alto.

A questo punto, si prosegue con l'applicazione dell'algoritmo di *clustering* che può essere effettuata in due modi:

- sfruttando la libreria MLlib che fornisce, tra i vari algoritmi per il Machine Learning, una variante parallelizzata di K-means++ [6], oppure
- sfruttando un'implementazione naive del K-means basata sull'algoritmo proposto da Lloyd [7].

6 Benchmark

La valutazione delle prestazioni è stata effettuata su sistema operativo Linux Ubuntu 20.04 virtualizzato con, a disposizione, 8 GB di RAM e 8 core (8 core di 16, Intel core i9 3,6 GHz - 5,0 GHz).

Per quanto riguarda l'esecuzione delle *query* si è notato che, eseguendo un *multi-run*, i tempi di esecuzione sono molto minori di quelli riscontrati nel singolo avvio. Questa differenza è dovuta all'istanziazione della JVM, nel tempo misurato per il singolo avvio è presente anche il tempo necessario per tale istanziazione, nel *multi-run*, invece, poiché la JVM viene istanziata solo al primo avvio, i tempi misurati nei successivi *run* sono indicativi esclusivamente del tempo di esecuzione dell'applicazione. Per portare in luce anche questa caratteristica dei tempi di esecuzione, i dati vengono divisi in misurazioni su *multi-start* e misurazioni su singola esecuzione. I tempi sono stati misurati senza considerare il *download* e l'*upload* dei dati da e verso l'HDFS; per eseguire una valutazione corretta, il *timer* viene fermato solo dopo aver ottenuto, nella macchina locale (o nel nodo *master* di Spark, in caso di istanziazione su *cluster*), il risultato finale (questo si traduce nell'inserimento di operazioni di `collect` anche laddove il solo salvataggio dell'RDD su HDFS fosse sufficiente per l'ottenimento di un output corretto). Al fine di garantire anche un fondamento statistico alle misurazioni effettuate, sono stati raccolti più campioni ed i dati presentati sono derivati eseguendo la media delle diverse osservazioni.

Funzionalità testata	**Tempo medio di esecuzione ± Deviazione standard (ms)**	
	Multi-start	**Single start**
Query 1 (Spark Core)	35,6667 ± 9,0779	1275,3333 ± 92,5304
Query 1 (Spark SQL)	775,2727 ± 24,5564	6229,0 ± 147,6144
Query 2 (Spark Core)	114,8788 ± 4,6187	1492,3333 ± 21,3417
Query 2 (Spark SQL)	1905,2727 ± 48,7015	8025,6667 ± 130,3912
Query 3 (Naive)	1399,5657 ± 48,1844	3705,6667 ± 54,8075
Query 3 (MLlib)	794,4040 ± 10,1789	2973,1667 ± 80,0585

Tabella 1: Benchmark

Per eseguire un confronto, nella terza *query*, tra la bontà della soluzione trovata usando l'implementazione naive di K-means e la bontà di quella trovata utilizzando MLlib, è stata utilizzata, come metrica, la *Within Set Sum of Squared Error* (WSSSE). Le soluzioni trovate dai due algoritmi, nella maggior parte dei casi, coincidono e, quando questo non avviene, si nota che, comunque, la WSSSE calcolata sul risultato dell'implementazione naive non si discosta mai di molto da quella calcolata sul risultato fornito da MLlib. Questo risultato permette di asserire la validità della soluzione naive in confronto a quella proposta dalla libreria MLlib.

Mese	**Somma delle WSSSE dei cluster individuati**	
	Naive	**MLlib**
01/2020	48,827821576648056	48,82782157664804
02/2020	4,2571098809791845	4,2190678645824695
03/2020	16343,19272628822	16343,192726288196
04/2020	4128,697916301173	4128,69791630117
05/2020	3820,8464520206717	3820,8464520206744

Tabella 2: Confronto algoritmi di clustering

REFERENCES

[1] https://github.com/pcm-dpc/COVID-19/blob/master/dati-andamento-nazionale/dpc-covid19-ita-andamento-nazionale.csv
[2] https://github.com/CSSEGISandData/COVID-19/blob/master/csse_covid_19_data/csse_covid_19_time_series/time_series_covid19_confirmed_global.csv
[3] https://craiggrummitt.com/2019/02/27/get-a-continent-from-longitude-latitude/
[4] https://nominatim.openstreetmap.org/
[5] https://www.bigdatacloud.com/
[6] http://theory.stanford.edu/~sergei/papers/vldb12-kmpar.pdf
[7] https://en.wikipedia.org/wiki/Lloyd%27s_algorithm