Analisi del dataset delle vaccinazioni anti Covid-19 con Apache Spark

Cecilia Calavaro
Corso di laurea magistrale
in Ingegneria Informatica,
Università degli studi di Roma
"Tor Vergata"
cecilia.calavaro@alumni.uniroma2.eu

***Abstract*—Sistema che risponde a delle queries riguardanti le vaccinazioni anti Covid-19 in Italia, utilizzando su una piattaforma standalone il framework di processamento Apache Spark ed il file system distribuito, portabile e scalabile di Hadoop (HDFS).**

1. Architettura di Sistema

E' stata realizzata un'applicazione per rispondere ad alcune queries riguardanti le vaccinazioni anti Covid-19 in Italia, utilizzando il framework di data processing Apache Spark. In particolare, il processamento dei dati è di tipo batch, ossia di dati già raggruppati e poi catalogati. I dati infatti sono in formato CSV; sono stati prelevati dalla repository GitHub "**https://github.com/italia/covid19-opendata-vaccini/tree/master/dati**" e sono stati importati nel sistema mantenendo tale formato. Secondo le specifiche di progetto, le queries effettuate sono la 1 e 2. Inoltre, si è risposto alla prima anche utilizzando SparkSql. La piattaforma utilizzata per il processamento delle queries è un nodo standalone su cui sono stati avviati due cluster: uno Spark per il processamento, ed un cluster HDFS per lo store dei dati sia di input che di output. In particolare, per la realizzazione dei due cluster è stato utilizzato **Docker Compose**, utile a definire ed eseguire applicazioni multi-container di tipo Docker. Tramite un file YAML, infatti, sono stati configurati i servizi dell'applicativo, che consistono in un'immagine che definisce il container per il master di spark, una per il worker, una per il namenode (nodo master) di hdfs ed una per il suo datanode (nodo worker). Inoltre, è possibile scalare i nodi worker dei due cluster andando a definire il numero di essi come parametro da riga di comando.

2. Scelte del design d'implementazione

L'applicazione è stata scritta nel linguaggio di programmazione Java ed è stata gestita con Apache **Maven**, in modo da facilitare il download automatico delle librerie necessarie al progetto e la risoluzione di eventuali dipendenze. In questo modo si è potuta raggiungere una facile portabilità del codice applicativo, sopratutto in termini di configurazione dell'ambiente. L'altro vantaggio nell'utilizzo di Maven consiste nella creazione di una **jar** del progetto che possa contenere tutte le dipendenze necessarie: quest'aspetto non è richiesto per l'attuale configurazione dell'applicazione, che al momento prevede solo l'uso Spark e hdfs, ma in un'eventuale implementazione futura, in cui verrebbero integrati più tools per i Big Data, tali dipendenze potrebbero essere indispensabili.
Facendo invece riferimento alla struttura della repository GitHub del progetto ("ceciliacal/sabd_project1"), il codice sorgente si trova nella cartella **"src/main/java"** ed è stato suddiviso in tre packages: "**queries**", contenente le classi "**Query1**" e "**Query2**", "**sparkSqlQueries**", che contiene la classe "**SqlQuery1**" ed infine "**utils**", avente delle classi con il solo scopo di supporto all'elaborazione degli outputs dell'applicazione.
Per quanto riguarda la previsione del numero di vaccinazioni nella query2, e quindi il calcolo della retta di regressione che approssima l'andamento delle vaccinazioni giornaliere, è stata utilizzata la libreria di **Apache Commons math3**, in particolare la classe **SimpleRegression**. Si è deciso di utilizzare quest'ultima piuttosto che Apache Spark MLlib in quanto si è riscontrato una maggiore facilità nell'utilizzo delle sue funzionalità.
D'altra parte, i files necessari al deployment dell'applicazione si trovano nella cartella "**docker**", che contiene il file **docker-compose.yml** e tutti gli scripts utili al deploy. Quest'ultimi sono stati codificati in bash e servono sia per il caricamento dei file csv di input su HDFS, sia per l'esecuzione dell'applicazione sul cluster di Spark. Nel file docker-compose.yml, le immagini dei containers Spark appartengono al servizio **Bitnami**, mentre quelle di HDFS sono state prelevate da **Big Data Europe**. La scelta dei due providers si è definita dopo svariati tentativi d'utilizzo di differenti Docker Images, in particolare per quanto riguarda Spark. Inizialmente infatti si era tentato di utilizzare, anche in questo, il servizio fornito da Big Data Europe e di conseguenza di seguire le direttive specificate nel loro README.md per quanto riguarda l'esecuzione di un'applicazione Java. Tuttavia si sono riscontrati numerosi problemi, e si è pertanto deciso di optare per Bitnami.
Si è scelto di utilizzare docker compose perché rappresenta una soluzione sia flessibile per quanto riguarda la scalabilità dell'applicazione, sia molto pratica grazie alla semplicità della configurazione del suo ambiente, poiché basta specificare ogni servizio necessario all'interno di un unico file. Il suo utilizzo infatti permette di configurare e avviare molteplici containers Docker sullo stesso host senza eseguirli separatamente. Inoltre, nonostante la piattaforma consista in un nodo standalone, l'approccio multi-container permette di simulare uno scenario di un sistema che lavori con i Big Data più realistico, in cui il processamento (o lo storage) venga suddiviso fra più macchine, che in questo caso sono rappresentate appunto da molteplici containers.

3. Descrizione dell'implementazione

Il codice sorgente per rispondere alle query 1 e 2 è stato suddiviso in diversi package, che sono descritti di seguito:

3.1. Package ”queries”

Le classi contenute in questo package sono:

- **StartQueries**: è la classe che contiene l’unico metodo ”main” dell’applicazione, in quanto ha lo scopo di avviarne il flusso logico quando va in esecuzione. In particolare, vengono computate in sequenza la query1, la query2 e la query1 elaborata con SparkSql, invocando, rispettivamente, i metodi ”Query1.query1Main()”, ”Query2.query2Main()” e ”SqlQuery1.computeQuery1()”.

  Degli aspetti comuni alle due classi Query1 e Query2 e che è bene specificare all’inizio sono i seguenti:

  1) Per misurare i tempi di processamento delle queries, tutte le classi appartenenti a questo package contengono un oggetto **Istant** che viene istanziato e avviato con il comando ”Istant.now()” subito dopo la creazione della Spark session e fermato, sempre con lo stesso comando, prima della chiusura di tale sessione.
  2) Poiché non è stata trasformata la rappresentazione dei dati in un altro formato durante la fase di data ingestion, i dati vengono direttamente letti dal file csv e splittati fra loro usando una virgola come separatore, per poi essere rappresentati con un JavaRDD di tipo stringa.

- **Query1**: questa classe implementa l’elaborazione della risposta alla prima query, la quale è suddivisa in due parti: la prima in cui attraverso il file **punti-somministrazione-tipologia.csv** bisogna calcolare il numero medio di centri vaccinali per regione, ed una seconda in cui, tramite il file **somministrazioni-vaccini-summary-latest.csv**, si vanno a calcolare il numero medio di vaccinazioni effettuate giornalmente da un generico centro vaccinale, per ogni mese e per ogni regione. Il metodo ”**query1Main()**” è quello che viene chiamato dalla classe ”StartQueries” per poter eseguire la prima query. All’interno di esso, come prima cosa viene creato un oggetto **SparkSession**, il punto d’accesso alle funzionalità di Spark e che è necessario a leggere i dataset di input dai files in formato csv che sono stati precedentemente caricati su HDFS. La fase di lettura dei dati avviene ovviamente due volte perché bisogna leggere da due files diversi; dopo la prima fase di lettura viene chiamato il metodo **computeNumVaccCentersPerArea**, e dopo la seconda **computeAvgVacc**. Il primo metodo serve per il calcolo del numero dei centri vaccinali per regione, mentre il secondo per quello del numero medio di vaccinazioni.
  Facendo un’analisi del metodo **computeNumVaccCentersPerArea**, dopo aver rimosso l’header, viene creato un JavaPairRDD che contiene i dati secondo i campi ”regione, denominazione struttura”. In questo modo è facile contare quante sono le strutture per regione, andando prima ad associare il numero ”1” ad ogni regione al posto di ognuna delle strutture (poiché nel JavaPairRDD ci saranno tante ripetizioni di una certa regione quanti sono le strutture diverse a essa associate, così come previsto dal dataset). Dopo viene sommato tale numero tante volte quante si ripete la regione tramite una trasformazione ”reduceByKey”, in modo da trovare il numero di centri vaccinali per ognuna di esse. In seguito a questa computazione viene invocato il metodo **computeAvgVacc**, che riceve come parametri di input il numero di centri vaccinali per regione (appena calcolato) ed il dataset di **somministrazioni-vaccini-summary-latest** trasformato in JavaRDD di tipo stringa. Una nota importante è che si è sempre lavorato tramite l’oggetto JavaPairRDD, quindi d’ora in avanti ”**RDD**” e ”**JavaPairRDD**” verranno usati come **sinonimi**. Anche in questo metodo, comunque, viene inizialmente rimosso l’header ed invocato un metodo di preprocessing **monthlyVaccinesPerArea**, il quale inserisce nel JavaPairRDD le colonne del dataset contenenti la data di somministrazione (anno-mese-giorno), la regione, ed il numero di vaccinazioni effettuate in tale regione quello specifico giorno. Sono considerati i dati compresi fra il 1 gennaio 2021 ed il 31 maggio 2021. Questo metodo, dunque, va a creare una tupla contenente la regione, il mese (estrapolato dalla data) ed il numero di vaccinazioni effettuate quel giorno, le quali vengono successivamente sommate in modo da ottenere le vaccinazioni totali effettuate in un certo mese in una specifica regione. Considerando la regione come chiave, viene effettuata poi una **join** fra questo RDD e quello contenente il numero di centri vaccinali per regione calcolato precedentemente dal metodo **computeNumVaccCentersPerArea** (che appunto, come già riportato, presenta i dati secondo ”regione, numeri centri vaccinali”). In seguito, viene calcolata la media del numero di vaccinazioni totali in un mese effettuate giornalmente in un generico centro vaccinale, andando a dividere il numero totale di vaccinazioni in un mese sia per il numero di centri vaccinali e sia per la cardinalità dei giorni di quel determinato mese. Infine, l’RDD risultante viene ordinato in modo ascendente per regione e per mese tramite la trasformazione ”sortByKey”che utilizza un **Comparator** di Tuple2. Il risultato finale viene posto in una lista di Tuple2 detta ”resultsList” tramite l’**azione** ”collect”. Tale lista viene poi scorsa invocando il metodo ”writeQuery1” della classe **CsvWriter** il quale, come suggerisce il nome, si occupa di scrivere l’output finale della query in formato csv sul file ”query1_result” che si trova nella cartella ”results”.

- **Query2**: in questa classe è presente il metodo principale chiamato ”query2Main()” che avvia la logica d’esecuzione per l’elaborazione del risultato della query2. Anche qui viene avviata la spark session che legge il dataset collocato nel file **somministrazioni-vaccini-latest.csv** ed inserisce i dati in un JavaRDD String, suddividendoli per virgola. Viene poi rimosso l’header ed effettuato un preprocessamento dei dati invocando il metodo **monthlyVaccines**, il quale preleva le colonne ”data_somministrazione” (contenente il giorno, il mese e l’anno), l’area (regione italiana), ”fascia_anagrafica” e ”sesso_femminile” (numero di vaccinazioni effettuate alle donne appartenenti ad una certa fascia anagrafica, in un determinato giorno ed in una specifica regione). Queste colonne vengono inserite in un RDD che possiede questo tipo di dati, fatta eccezione per la data di somministrazione che diventa il primo del mese di quella data. In seguito viene rimossa la distinzione secondo la casa farmaceutica che ha fornito i vaccini

inoculati in uno specifico giorno, andando a sommare tutte le tuple aventi stessa chiave (mese, regione, fascia anagrafica).
Come viene richiesto da specifica, vengono quindi considerate soltanto le fasce anagrafiche per cui in quel mese vengono registrati almeno due giorni di campagna vaccinale. Per effettuare questo filtrazione dei dati, viene invocato il metodo **checkMoreThan2DaysPerMonth**, in cui viene creato un oggetto Map avente come **key** "Mese,Regione,Fascia anagrafica" e come **value** la cardinalità dei giorni in cui sono stati effettuati vaccini in quel mese. Quindi viene effettuata una trasformazione **filter** sull'RDD che manterrà solamente le tuple la cui chiave (sempre "Mese,Regione,Fascia", la stessa utilizzata nella Map) è mantenuta nel dictionary.
Una volta attuati questi controlli sui dati nell'RDD con cui si sta lavorando, è possibile andare a calcolare la predizione del numero di vaccinazioni che saranno fatte il primo giorno del mese successivo. Per realizzare la predizione è stata utilizzata la classe SimpleRegression della libreria di Apache Commons math3. Tale predizione viene calcolata invocando il metodo **calculateSortedPredictions**. Questo metodo ne invoca a sua volta un altro, chiamato **calculateRegression** che va a creare un JavaPairRDD chiamato "**month_area_age_SimpleReg**" che, come suggerisce anche il nome, è composto da una Tupla3, identificabile come una chiave, formata da "mese, regione, fascia anagrafica" e da un oggetto SimpleRegression come valore. Ad ogni istanza di SimpleRegression degli elementi dell'RDD aventi stessa chiave (e ci saranno tanti elementi con stessa chiave quante sono le date appartenenti ad uno stesso mese, per ogni fascia anagrafica e per ogni regione) viene aggiunto il giorno ed il numero di vaccinazioni effettuate in quello stesso giorno tramite il metodo "**SimpleRegression.addData(x,y)**", dove appunto la x è il giorno e la y coincide con il numero di vaccini. Una volta restituito l'RDD "month_area_age_SimpleReg" al cui oggetto SimpleRegression sono stati aggiunti i valori del giorno e del numero di vaccini, viene effettuata su di esso una trasformazione "**reduceByKey**" in cui viene effettuato un'operazione di "**append(SimpleRegression reg)**". Questo metodo appartiene alla libreria Apache Commons math3, e appende i dati dal calcolo di un'altra regressione lineare all'istanza SimpleRegression chiamante. Con la **reduceByKey** quindi viene effettuata un'operazione di append su tutti gli oggetti SimpleRegression aventi stessa chiave. In altre parole, ad ogni chiave "mese, regione, fascia anagafica" corrisponderà un solo oggetto SimpleRegression, che contiene tutti i punti x e y intesi rispettivamente come il numero del giorno in un certo mese ed il numero di vaccinazioni fatte in quel preciso giorno, per tale regione e fascia d'età. Infine, il metodo "**calculateSortedPredictions**" invoca un ulteriore metodo, chiamato "**predictFirstDayOfNextMonth**" in cui viene effettuata la predizione vera e propria del numero di vaccinazioni per il mese successivo, per ogni regione e per ogni fascia d'età, in base ai dati del mese precedente. In particolare, qui viene utilizzata la funzione "**SimpleRegression.predict(x)**" che ritorna il valore predetto "y" associato al valore "x" fornito, sulla base dei dati che sono stati precedentemente aggiunti al modello tramite le funzioni "addData(x,y)" e "append(SimpleRegression reg)". Poiché le x del modello s'identificano numericamente con i giorni di un mese, il valore del giorno che viene passato in input al metodo "SimpleRegression.predict(x)" coincide con l'ultimo giorno del mese aumentato di uno. Per fare un **esempio pratico**: il mese di febbraio ha 28 giorni, per calcolare il numero di vaccinazioni predette per l'1 Marzo viene passato il numero x=29, perché appunto 29 è il numero successivo a 28, che è appunto l'ultimo giorno.
Gli elementi del JavaPairRDD vengono poi ordinati per chiave ed infine viene calcolata la classifica delle regioni in cui è previsto il maggior numero di vaccinazioni il primo giorno del mese successivo, per ogni fascia anagrafica e per ogni mese. Questa classifica viene computata nel metodo "**sortPredictionsByKey**", che infine restituisce una lista di Tuple2, la quale viene riportata dal metodo writeQuery2 nel file di outpt "query2_result".

3.2. Package "sparkSqlQUeries"

- **SqlQuery1**: questa classe ha un metodo principale, che viene invocato dalla classe queries.StartQueries all'avvio dell'applicazione, chiamato "computeQuery1". Lo scopo è quello di rispondere alla query1 usando il linguaggio SparkSql: parallelamente a ciò che accadeva nella classe queries.Query1, vengono invocati due metodi che suddividono l'elaborazione del codice sorgente in base ai due files che bisogna utilizzare (**punti-somministrazione-tipologia.csv** e **somministrazioni-vaccini-summary-latest.csv**). Dopo aver inizializzato gli oggetti **SparkSession** e **Istant** per valutare il tempo d'esecuzione della query, viene chiamato il metodo "**preprocessNumCenters**", il quale crea un **Dataset** di tipo **Row** dal file csv "punti-somministrazione-tipologia" e conta il numero di centri vaccinali per regione tramite una query in linguaggio SQL, il linguaggio standard per comunicare con i database di tipo relazionale. In seguito viene invocato il metodo "**preprocessVaccinesSomm**" per il calcolo del numero medio di vaccinazioni, all'interno di cui, dopo aver letto i dati dal relativo csv nella stessa modalità illustrata precedentemente, vengono effettuate **tre queries** SQL:
  - una per selezionare la regione, la data ed il numero di vaccinazioni comprese fra il 1 gennaio 2021 ed il 31-05-2021;
  - una che effettua la join con il dataset avente la regione ed il numero di centri vaccinali elaborato nel metodo "preprocessNumCenters" e che restituisce un dataset che abbia anche una colonna per il mese (al posto della data giornaliera) ed una per il numero di giorni presente in quel mese (utile per effettuare successivamente la divisione nel calcolo della media);
  - un'ultima query che restituisce per ogni regione e mese, il numero medio di vaccinazioni giornaliere effettuate in un generico centro vaccinale di quella regione.

3.3. docker

L'avvio dei due cluster può essere automatizzato facendo il run solamente dello script "**run.sh**" presente in questa cartella. Infatti, esso esegue, in ordine:
-**start_docker.sh**: avvia i due cluster tramite il comando "**docker-compose up**" in modalità detached permettendo di specificare come parametri il numero di nodi workers sia

per spark che per HDFS. Dopo questo script, in ”run.sh” è presente una **sleep** di 40 secondi per assicurarsi che il cluster sia già avviato prima che vengano effettuate operazioni che lo coinvolgano;
-**uploadCsv.sh**: vengono scaricati i tre csv utilizzati per rispondere alle queries direttamente dal loro URL, in questo modo ogni volta che l'applicazione verrà eseguita i dati saranno sempre aggiornati. Quest'aspetto non presenta un problema perché nelle query le date utilizzate sono comprese all'interno di un certo intervallo, come specificato nei requisiti progettuali. I files vengono poi caricati sul namenode di HDFS;
-**runQueries.sh**: viene effettuato lo **spark-submit** dell'applicazione, e quindi eseguita la classe queries.StartQueries sul cluster Spark.

Per effettuare il **deployment** quindi è sufficiente eseguire i seguenti comandi:
- ”**mvn package**” per la creazione del file jar
- ”**./run.sh**” nella cartella ”docker” (in riferimento alla repository GitHub)

4. Limitazioni

- Nel codice sorgente non sono stati utilizzati sugli RDD i metodi **persist()** o **cache()**, i quali permettono di effettuare lo store in memory del dataset durante le molteplici operazioni previste dall'applicazione: ogni nodo memorizzerebbe le sue partizioni per riutilizzarle in successive azioni sullo stesso dataset. Il motivo per cui non sono stati utilizzati questi metodi è che i dataset utilizzati in questo progetto non di piccole dimensioni, infatti due dei files di input hanno circa 3000 righe e solo uno circa 60.000. Non si parla quindi di dataset di grandi dimensioni, aventi numeri che raggiungono l'ordine delle centinaia di migliaia o di milioni di righe; pertanto nel caso in cui avvenga un guasto non sarebbe dispendioso rielaborare le operazioni nuovamente. Tuttavia, i metodi persist() o cache() consentono anche di velocizzare azioni future e di offrire una tolleranza ai guasti, due aspetti che in un approccio più realistico sarebbero sicuramente necessari.
- La funzionalità della classe utils.CsvWriter è di scrivere i risultati delle queries sui due files csv di output, ”query1_result.csv” e ”query2_result.csv”. Poiché alla fine del processamento di entrambi le queries si hanno due liste di tuple composte da **key** e **value**, nell'implementazione di tale classe sono state lasciate delle parentesi che racchiudono la chiave, che inoltre è stata separata dal relativo value da un punto e virgola (”;”). Il motivo di questa scelta è dettato dal fatto che se si visualizzano i files in formato csv su excel, la key viene collocata sulla prima colonna ed il value in quella accanto, in modo da facilitarne la lettura e rendere l'output più chiaro.

5. Piattaforme Software

Il sistema è stato realizzato tramite:
- **IntelliJ IDEA Ultimate Edition** come IDE
- Apache **Maven** 3.6.0
- Macchina Virtuale con **Ubuntu 18.04**
- **docker** 20.10.6
- **docker-compose** 1.17.1

6. Risultati dei Tempi d'Esecuzione

I test sono stati svolti utilizzando un processore i7-7700HQ. In particolare, come si può vedere anche nella tabella riportata nella **Figure 1**, sono stati effettuati 3 cicli di test, aventi un numero di nodi worker per hdfs e per spark pari a 2, 3 e 4. Il tempo è stato misurato in millisecondi (ms)

hdfs	spark	query1	query1SQL	query2
n	n	ms	ms	ms
2	2	9.766	11.136	3.917
3	3	10.294	11.576	3.929
4	4	10.513	12.618	5.175

Figure 1. Tabella

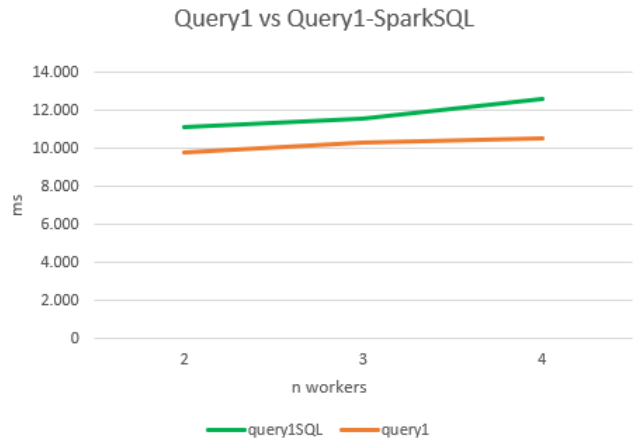

Figure 2. Grafico query1 vs query1-sparkSQL

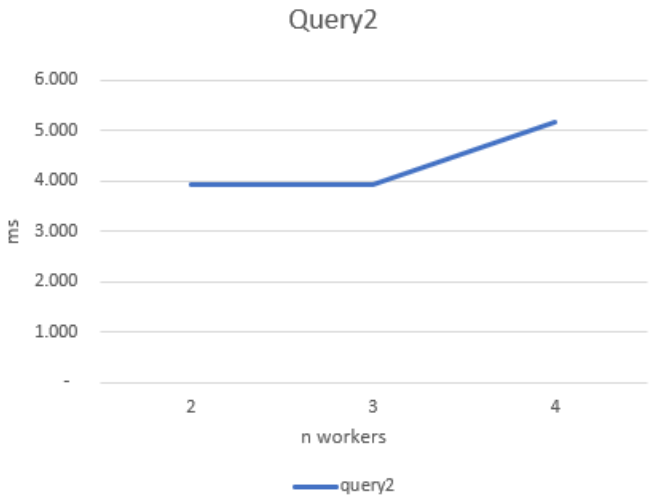

Figure 3. Grafico query2

7. Conclusione

Come si può osservare anche dai grafici riportati in **Figure 2** e in **Figure 3**, l'analisi è stata effettuata con diversi nodi worker, ma poiché i dataset utilizzati sono di piccole dimensioni non si notano differenze significative al variare dei nodi, sia in termini d'aumento che di diminuizione. Ci si sarebbe aspettato

un notevole aumento delle prestazioni se fosse stato utilizzato un dataset di dimensioni più grandi oppure su un sistema distribuito vero e proprio, che non sia un nodo standalone. Al contrario, dai grafici addirittura si può evincere come all'aumentare del numero di nodi il tempo d'esecuzione delle query comunque aumenti. Questo si può giustificare con il fatto che essendo appunto un dataset di piccole dimensioni, l'overhead impiegato per l'allocamento delle risorse è maggiore rispetto al tempo di computazione vero e proprio dell'applicazione. Inoltre, si può notare come SparkSql non sia una soluzione conveniente rispetto all'utilizzo degli RDD, in quanto prevede tempi di processamento maggiori.

References

[1] libreria "Apache commons math3 SimpleRegression": "https://commons.apache.org/proper/commons-math/javadocs/api-3.3/org/apache/commons/math3/stat/regression/SimpleRegression.html"

[2] Big Data Europe Hadoop: "https://github.com/big-data-europe/docker-hadoop"

[3] Bitnami Spark: "https://github.com/bitnami/bitnami-docker-spark"