UNIVERSITÀ
POLITECNICA
DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

---

BVQ Spark - Ottimizzazione ed Estensione

Abbadini Lorenzo - 1088286
Moscoloni Elena - 1085408
Scisci Valerio - 1088283

ANNO ACCADEMICO 2019/2020

Indice

1. Introduzione

Con questo progetto si intende andare ad integrare ed espandere il lavoro portato a termine da un altro gruppo di ragazzi, che si sono occupati della realizzazione dell'algoritmo **BVQ** nel linguaggio di programmazione **Java** tale da funzionare sulla piattaforma **Spark**.

Allo stato attuale, l'implementazione fornitaci dell'algoritmo prevede i seguenti passi:

- Divisione dei dati in un blocco di training e uno di test in formato **RDD** funzionale all'esecuzione in Spark;
- In base al numero di vettori codici scelto e al numero di classi presenti, viene creata una rete neurale e la relativa matrice dei pesi iniziale;
- Vi è la possibilità o meno (impostando un flag a true/false) di eseguire una prima fase di addestramento usando una **SOM**(quindi un'altra rete neurale), che andrà ad aggiornare i pesi in modo inversamente proporzionale al tempo in base al valore di **Learning Rate** scelto;
- A questo punto viene addestrata la prima rete neurale creata, quella che si occuperà di eseguire l'algoritmo vero e proprio del BVQ;
- Fase di testing dove viene usata la rete per individuare la classe di appartenenza dei dati contenuti nel test set, individuando anche gli elementi miss-classificati;
- Calcolo di matrice di confusione e altre metriche per misurare la qualità della classificazione effettuata;
- Il tutto viene salvato in un file csv e in un file excel;
- Nota 1: durante la fase di training delle due reti neurali, dove verranno modificati i pesi in base agli input del training set, in base al valore del flag "**replicabile**", si hanno tre possibili evoluzioni:
    - "**false**", esperimento non replicabile e diverso per ogni esecuzione, ma viene sfruttata al massimo la parallelizzazione di Spark. I punti vengono presi a blocchetti di dimensione **dim_bag** o viene preso l'intero training_set (in base al valore del flag "flag_iteraz" e uno alla volta vengono usati per allenare la rete). La non replicabilità è dovuta al diverso ordine di terminazione sui diversi worker dove spark distribuisce l'esecuzione;
    - "**true**", viene eseguito tutto il lavoro su un solo worker, eliminando il parallelismo, ma permettendo la replicabilità dell'esperimento. In sostanza viene lanciata una **collect** che accentrerà i dati e dunque l'esecuzione che non risulterà più randomica;
    - "**secure**", l'esperimento può essere ripetuto, e viene garantito il concetto di epoca. Inoltre, i punti dell'intero training_set vengono presi uno alla volta e questo causa un aumento notevole dei tempi di esecuzione.
- Nota 2: nell'implementazione attuale vengono garantiti e implementati il concetto di epoca SOM/BVQ e quello di iterazione BVQ.

Gli obiettivi che ci siamo posti sono quindi i seguenti:

- Migliorare l'efficienza, se possibile;
- Verificare se esiste un modo per rendere effettivamente parallelizzata l'esecuzione dell'algoritmo mantenendo allo stesso tempo la replicabilità degli esperimenti;

- Realizzare una serie di esperimenti per valutare i risultati ottenuti nelle precedenti fasi e per testare la scalabilità dell'algoritmo.

2. Miglioramento dell'efficienza

In questa sezione vengono discussi i miglioramenti relativi all'efficienza del programma che sono stati effettuati; essi riguardano sia miglioramenti del codice, come la riduzione di istruzioni utilizzate a parità di funzionalità o l'eliminazione di codice superfluo, sia soprattutto il miglior sfruttamento delle tecnologie offerte da Spark.

2.1 Passaggio da RDD a DataFrame

La prima modifica che è stata apportata al codice riguarda l'utilizzo della struttura dati **DataFrame** al posto del RDD.

2.1.1 RDD e Spark

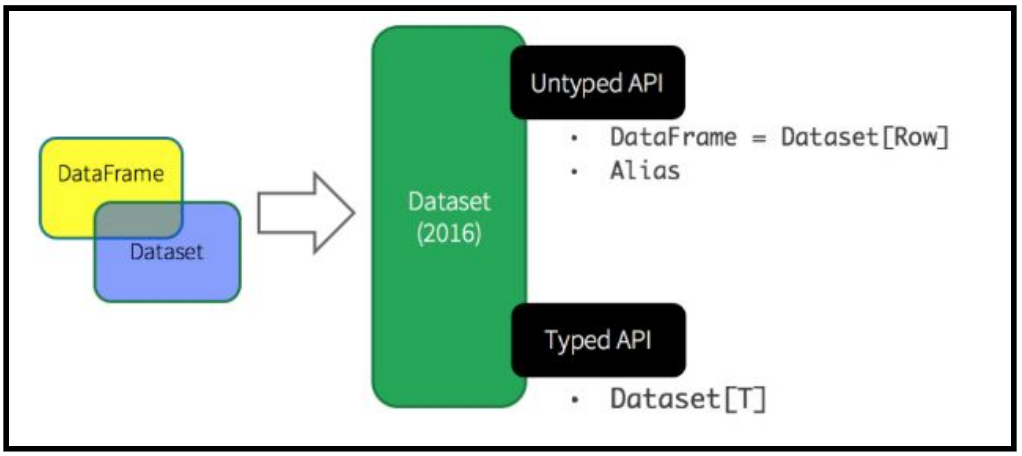
L'RDD, Resilient Distributed Dataset, rappresenta la principale API messa a disposizione dalla piattaforma Spark. Un RDD non è altro che un insieme immutabile di elementi distribuiti, sui quali è possibile agire in modo parallelo tramite un'API di basso livello fornita dalla stessa piattaforma. L'uso di un RDD è consigliato nei seguenti casi:

- ➢ si vogliono attuare trasformazioni di basso livello sui dati;
- ➢ i dati non sono strutturati;
- ➢ non è importante fornire uno schema formale ai dati, come l'organizzazione in colonna degli attributi, durante le fasi di elaborazione o di accesso ai valori;
- ➢ si è disposti a rinunciare ai vantaggi messi a disposizione dai DataFrames e Dataset in termini di ottimizzazione.

2.1.2 DataFrame/Dataset e Spark 2.0

Similmente ad un RDD, un DataFrame è una raccolta immutabile di dati distribuiti, che risultano però ben organizzati in una serie di colonne caratterizzate ognuna da un nome che ne indica la semantica, come accade in una tabella di un database relazionale.
L'elaborazione di tali dati risulta molto più accessibile grazie ad un'astrazione che avviene ad un livello superiore: è presente infatti un'apposita API dedicata alla gestione di questi dati distribuiti.

In Apache Spark 2.0 i DataFrame e i DataSet di dati sono unificati a rappresentare la stessa struttura e vengono pertanto gestiti come fossero un'unica API che prende il nome di Dataset, come ci mostra la seguente immagine.



In particolare, un DataSet di dati presenta due tipologie di API: una fortemente tipizzata, dove viene definito in modo categorico il tipo di dati che viene salvato al suo interno, e un'altra non tipizzata, che salva i dati al suo interno in modo generico come una riga (Row) di una tabella.
Possiamo considerare i DataFrame come un alias di una generica collezione di oggetti organizzati in un Dataset; ciò che distingue i primi dai secondi consiste nel fatto che i primi non specificano la tipologia dei dati che raccolgono, rappresentando dunque l'API non tipizzata, viceversa i secondi sono una collezione di oggetti fortemente tipizzati

Language	Main Abstraction
Scala	Dataset[T] & DataFrame (alias for Dataset[Row])
Java	Dataset[T]
Python*	DataFrame
R*	DataFrame

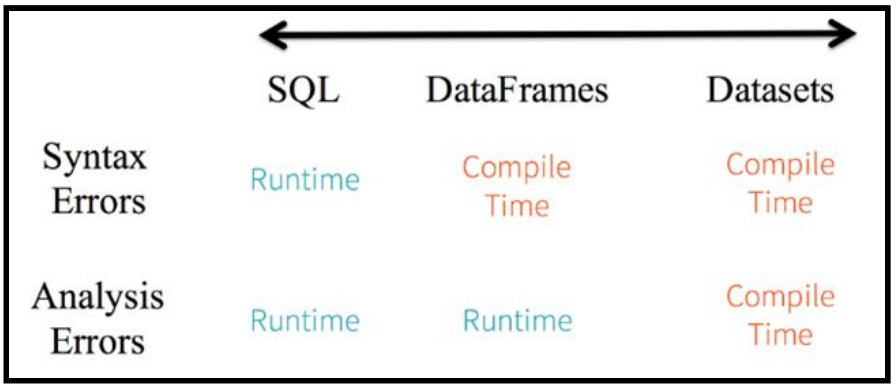
Supporto delle astrazioni nei vari linguaggi

2.1.3 Vantaggi nell'utilizzo dei DataFrame/Dataset

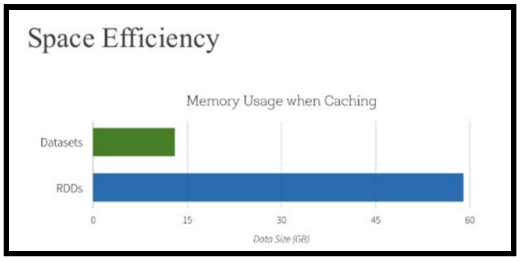
- In presenza di DataFrame e Dataset, gli errori di sintassi vengono rilevati durante la fase di compilazione e ciò permette di risparmiare tempo e costi.

  Se viene, per esempio, richiamata una funzione che non fa parte dell'API di tipo DataFrame, questo errore verrà catturato dal compilatore, tuttavia quest'ultimo non sarà in grado di rilevare un eventuale nome di colonna errato fino alla fase di runtime.

  Per un'API di tipo Dataset, qualsiasi mancata corrispondenza dei parametri tipizzati sarà rilevata in fase di compilazione, essendo tale struttura dati più restrittiva;



- I DataFrame permettono di realizzare un'astrazione di alto livello che consente di rappresentare i dati in modo strutturato o semi-strutturato, tramite ad esempio le classi di Scala;
- La presenza di un tipo di dato ben strutturato, sebbene possa limitare l'uso che se ne può fare, garantisce comunque una semantica ricca e un semplice set di operazioni esprimibili come costrutti di alto livello. L'utilizzo di queste API risulta pertanto facilitato rispetto all'RDD;
- Infine queste strutture garantiscono maggiore efficienza in termini di memoria e migliori prestazioni, innanzitutto perché tali API si basano sul motore Spark SQL e pertanto tutte le query sono soggette allo stesso ottimizzatore, anche perché **Tungsten*** è in grado di serializzare/deserializzare (serializzare un oggetto vuol dire convertirlo in un flusso di bytes) in modo efficiente gli oggetti JVM generando un bytecode che può essere eseguito a velocità superiori. In sostanza, la serializzazione che avviene per gli oggetti di tipo RDD, in particolare con Java, è poco efficiente e genera molto overhead soprattutto in fase di distribuzione da parte di Spark tra i Worker di oggetti di questo tipo.



Il loro utilizzo è dunque consigliato quando:

- Si desidera avere una semantica ricca e astrazioni di alto livello;
- L'elaborazione richiede l'uso di funzioni di filtraggio, mappatura, aggregazione, media, somma, query SQL e accesso mediante i campi delle colonne;
- È necessaria maggiore sicurezza in fase di compilazione;
- Si voglia sfruttare la maggiore efficienza e garantire elevata ottimizzazione grazie all'ottimizzatore **Catalyst****;
- Si voglia optare per un approccio più semplice e accessibile.

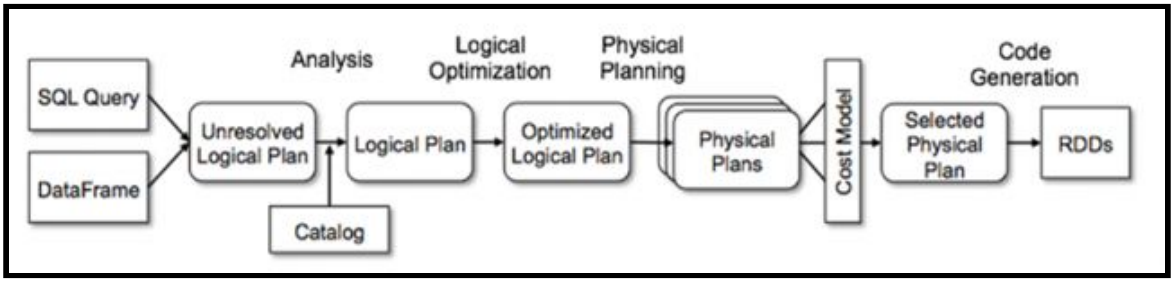
*Tungsten è il progetto realizzato con lo scopo di apportare modifiche al motore di esecuzione di Apache e si focalizza sul miglioramento dell'efficienza in termini di memoria e di CPU per le applicazioni Spark.

Tale progetto include:

- ❏ La gestione della memoria e l'elaborazione binaria. In particolar modo si sfrutta la semantica delle applicazioni per la gestione esplicita della memoria e si elimina il sovraccarico generato dal modello ad oggetti JVM e dalla garbage collection;
- ❏ Algoritmi e strutture dati per sfruttare la gerarchia della memoria cache;
- ❏ L'utilizzo della generazione di codice per sfruttare compilatori e CPU moderni;
- ❏ L'assenza di funzioni virtuali che riduce le chiamate alla CPU, le quali potrebbero causare un forte abbattimento delle prestazioni;
- ❏ L'inserimento dei dati nei registri della CPU. Ciò permette di ridurre il numero di cicli poiché i dati vengono ripescati dai registri della CPU anziché dalla memoria;
- ❏ La compilazione e l'esecuzione di semplici loop al posto di funzioni complesse.

** Catalyst è l'ottimizzatore di Spark SQL che sfrutta delle funzionalità avanzate del linguaggio di programmazione per creare un ottimizzatore di query estensibile. Si basa sui costrutti di programmazione funzionale di Scala e si pone i seguenti obiettivi:

1) Aggiungere a Spark SQL nuove tecniche e funzionalità per incrementarne l'ottimizzazione;
2) Consentire anche agli stessi sviluppatori esterni di ampliare tale ottimizzazione tramite l'aggiunta di nuove regole.



Come si nota da questa immagine, i DataFrame in realtà a livello fisico vengono trattati esattamente nello stesso modo degli RDD. Il motore però si occupa di effettuare tutta una serie di ottimizzazioni.

2.1.4 Passaggio dal solo Spark Context alla Spark Session

Per poter utilizzare i DataFrame è necessario sfruttare le funzionalità SQL di Spark; ciò comporta il passaggio diretto dallo **Spark Context** alla **Spark Session**.

La Spark Session fornisce la possibilità di integrare le varie funzionalità di Spark sfruttando un numero minore di costrutti: invece di avere un context Spark, un context Hive, un context SQL, ora tutti questi elementi sono incapsulati in una Session Spark

Prima dell'avvento di Spark 2.0, Spark Context rappresentava il punto di partenza di qualsiasi applicazione Spark e veniva utilizzato per accedere a tutte le sue funzionalità. A partire da Spark Context è stata implementata la struttura RDD e in seguito è stata necessaria la creazione di

specifici Context adatti alle singole interazioni di spark: SQLContext per SQL, HiveContext per hive, Streaming Application per lo streaming ecc..

Spark Session si pone l'obiettivo di combinare tutti questi diversi contesti ai quali è possibile accedere tramite l'unico oggetto SparkSession.

Questo permette allo sviluppatore di non doversi preoccupare di dover creare diversi contesti; inoltre facilita anche l'interazione di diversi utenti nello stesso contesto Spark.

Per poter creare una Spark Session bisogna eseguire il seguente builder pattern:

```
import org.apache.spark.sql.SparkSession
SparkSession spark_session = SparkSession.builder()
.appName("SparkSessionExample")
.master("local[4]")
.getOrCreate();
```

Di seguito è riportato uno screen per evidenziare le modifiche apportate al codice:

Come è possibile notare dalle righe di codice della classe MainClass aggiornata, per poter convertire i dati dal formato libsvm in DataFrame, invece che in JavaRDD, è stato necessario utilizzare la funzione dedicata:

```
data_spark = spark_session.read()
        .format("libsvm")
        .option("numFeatures", 2)
        .load(path);
```

Il DataFrame così creato presenta due colonne: la prima, nominata "label", che contiene un valore double ad indicare la classe, la seconda, nominata "features", che contiene i dati in formato **SparseVector**.

2.1.5 Passaggio da LabeledPoint a <Row>

Inizialmente i singoli elementi da sottoporre a classificazione venivano salvati in una variabile di tipo **LabeledPoint** (una struttura dati che identificava un oggetto come la sua classe di appartenenza, label, e le sue caratteristiche, feature), ciò come conseguenza dell'utilizzo della libreria MlLib con gli RDD.
Passando ai DataFrame, gli elementi da classificare non sono più LabeledPoint ma singole righe della nuova struttura dati. Tale passaggio ha causato una serie di problemi relativi alla conversione di tipo, in quanto ad esempio nei LabeledPoint le features sono salvate come Vector e il metodo "feature()" permette di accedere in automatico al valore della feature, mentre nei DataFrame le features sono salvate come SparseVector e i singoli valori contenuti all'interno delle sue righe devono essere estratti manualmente.

2.1.6 Verifica Migliorie Apportate

A questo punto si sono eseguite alcune prove per mettere a confronto (a parità di dati) le differenze a livello di efficienza che si sono ottenute.

Si è preso un campione di dati con due feature e appartenenti a due possibili classi, composto di 10000 elementi (5000 per ogni classe).

Le prime misure empiriche ottenute sono le seguenti:

- La creazione della Spark Session impiega in media 0.45 secondi, mentre la creazione dello Spark Context che si usava in precedenza impiegava mediamente 0.06 secondi. Il risultato è coerente con ciò che ci aspettiamo dato che la Spark Session integra molte più funzionalità come sopra descritto;
- La fase di caricamento dei dati e poi di splitting in *training_set* e *test_set* è più onerosa e la spiegazione più logica è che i dati sono strutturati a differenza di quando si usavano gli RDD, richiedendo quindi più tempo per la costruzione della struttura dati in sé. In particolare, il tempo di esecuzione per questa fase del programma è all'incirca 3 secondi, 2 per il caricamento e 1 per lo splitting, mentre prima erano necessari 1,5/2 secondi in tutto. Risulta chiaro che questi tempi scaleranno proporzionalmente alla dimensione dei dati da caricare.

A questo punto abbiamo messo a confronto i tempi di esecuzione delle due fasi dell'algoritmo vero e proprio, cioè l'allenamento della SOM e l'esecuzione del BVQ stesso.
Si sono mediati i risultati di accuratezza e tempi di esecuzione di 10 prove per ogni configurazione dei parametri iniziali scelta.

Le prove prevedevano i seguenti parametri:

1. Prima prova: [Replicable = false, Epoche = 1, SOM = true, Iterazioni BVQ = 10, dim_bag=100];
2. Seconda prova: [Replicable = false, Epoche = 1, SOM = false, Iterazioni BVQ = 10, dim_bag=100];

3. Terza prova: [Replicable = false, Epoche = 5, SOM = true, Iterazioni BVQ = 10, dim_bag=100];
4. Quarta prova: [Replicable = false, Epoche = 5, SOM = false, Iterazioni BVQ = 10. dim_bag=100].

	Vecchia Versione (RDD)	**Nuova Versione (DataFrame)**
Prima Prova	SOM: 0,64 s BVQ: 1,075 s Accuratezza: 0,6427	SOM: 0,34 s BVQ: 1,2435 s Accuratezza: 0,6093
Seconda Prova	SOM: --- BVQ: 1,07 s Accuratezza: 0,6364	SOM: --- BVQ: 1,156 s Accuratezza: 0,5882
Terza Prova	SOM: 0,205 s BVQ: 1,238 s Accuratezza: 0,6589	SOM: 0,591 s BVQ: 1,065 s Accuratezza: 0,6721
Quarta Prova	SOM: --- BVQ: 1,1695 s Accuratezza: 0,6343	SOM: --- BVQ: 1,106 s Accuratezza: 0.5861

Dai risultati appena elencati in realtà non emerge un vincitore assoluto tra le due versioni. Si osserva mediamente un aumento, in alcuni casi anche considerevole, della velocità di esecuzione del BVQ, mentre relativamente alla SOM i cambiamenti dei tempi potrebbero essere dovuti anche alle leggermente diverse condizioni di esecuzione dei test.
Per quanto riguarda l'accuratezza siamo nel range di variazione naturale che si ha con l'algoritmo, data la non ripetibilità dell'esperimento.

Nel prossimo capitolo viene presentata però un'ulteriore motivazione per la quale la scelta della versione basata su DataFrame dell'algoritmo potrebbe essere conveniente: l'aggiunta di una colonna "id" per rendere deterministica la scelta dei punti in fase di training.

3. Replicabilità e Parallelismo

Come detto nell'introduzione, ad ora non esiste un modo per rendere replicabile l'algoritmo se il calcolo viene distribuito parallelamente tra i vari nodi. Quello che succede infatti è che l'ordine di terminazione dell'elaborazione nei vari nodi è casuale e non è possibile controllarlo. L'unico modo è accentrare i dati e usare un seed come hanno già fatto i ragazzi prima di noi.

Il passaggio alla struttura DataFrame ha però aperto le porte ad una possibile soluzione: verrà aggiunta una colonna "id" che permetterà di identificare univocamente le righe e quindi i punti con cui vogliamo allenare/testare la rete neurale. Sarà possibile, sempre sfruttando un seed, andare a prendere i punti con un certo ordine che sia tale per ogni esecuzione a prescindere dalla

distribuzione fisica dei dati tra i vari worker di Spark. In teoria questa cosa dovrebbe garantire la ripetibilità dell'esperimento.

3.1 Inserimento della Nuova Colonna

Data la nuova necessità venutasi a creare, si è preferito passare ad un caricamento dei dati a partire da un file csv.
Prima di questo momento i dati venivano caricati da un file .txt dove i punti erano salvati in formato libsm e poi erano convertiti in modo automatico in un DataFrame tramite la funzione descritta nel paragrafo 2.1.4 ottenendo la conversione seguente:

file.txt		**DataFrame**
[Classe feature1:valore_feature feature2:valore_feature]	→	*[Classe | SparseVector]*

Adesso il caricamento avverrà direttamente da un file csv dove i dati saranno strutturati nel seguente modo:

[id,Classe,valore_feature_1,valore_feature_2]

Tramite il codice seguente verrà effettuata la conversione in DataFrame a partire dal file.csv:

```
data_spark = spark_session
        .read()
        .schema(schema)
        .csv(path).cache();
```

```
+---+-----+--------------------+--------------------+
| id|label|            feature1|            feature2|
+---+-----+--------------------+--------------------+
|  1|  1.0|  0.3308221032873145|  0.9742723791313979|
|  2|  1.0| 0.10964331645076425| -0.7316892032898228|
|  3|  1.0|  -1.521017474559067|-0.08347837202016653|
|  4|  1.0| 0.09971783324666713|  0.7014729813013303|
```

DataFrame

3.2 Adattamento dell'Algoritmo

A questo punto bisogna adattare l'algoritmo in modo da sfruttare la nuova colonna aggiunta per renderlo deterministico anche quando parallelizzato.
Per attivare le nuove funzionalità è stato previsto un nuovo valore del flag "**replicabile**" ovvero "**secure_index**". Il parametro andrà quindi settato come di consueto tramite il file "**comando_di_lancio.txt**".

Le funzioni che sono intaccate dalla modifica sono quelle che selezionano i punti per allenare la SOM e la rete della BVQ. Esse dovranno andare a prendere gli elementi dal training set sempre nello stesso ordine. Per farlo sfruttiamo la colonna "id": prendiamo la lista di id dei punti appartenenti al training_set (che precedentemente sono stati selezionati in modo randomico in base al seed) per poi andarli a prendere uno alla volta sempre nello stesso ordine a parità di seed. Così facendo la fase di training sarà di volta in volta identica. Viene quindi riportato il nuovo codice e poi discusso meglio in dettaglio.

Nelle immagini si può vedere il codice che fa le seguenti operazioni:

1. Se il flag_index è true prende dim_bag elementi per num_bag volte altrimenti prende tutto il training_set;
2. Prendo la lista di ID dei punti appena selezionati che userò per fare training;
3. Si prendono i punti uno alla volta selezionandoli tramite il loro id e si esegue l'allenamento.

```
case "secure_index":
    if(flag_iteraz){

        for(int i=0; i<=num_bag; i++){
            Dataset<org.apache.spark.sql.Row> p = pset.sample(true, 1D*dim_bag/dim_pset, seed+i).limit(dim_bag);
            p.cache();
            List<org.apache.spark.sql.Row> IDs;
            IDs = p.select("id").collectAsList();
            for(int j=0; j < IDs.size(); j++) {
                learn(p.where("id='"+IDs.get(j).getInt(0)+"'").first());
            }
        }
    }else{
        List<org.apache.spark.sql.Row> IDs;
        IDs = pset.select("id").collectAsList();
        pset.cache();
        for(int i=0; i<epoch; i++){
            for(int j=0; j<IDs.size(); j++) {
                learn(pset.where("id='"+IDs.get(j).getInt(0)+"'").first());
            }
        }
    }
    break;
```

Implementazione estrazione punti train SOM.

Nella seconda immagine è riportato il codice che stabilisce l'ordine di training del BVQ.
Esso si comporta esattamente come il primo e prende i punti in maniera deterministica a parità di seed. L'unica differenza è che converte i punti in formato vettoriale double con la funzione PointToDoubleVec2 per poter usare la funzione di learn del BVQ.

```
case "secure_index":
    if(flag_iteraz){

        for(int i=0; i<=num_bag; i++){
            Dataset<org.apache.spark.sql.Row> p = pset.sample(true, 1D*dim_bag/dim_pset, seed+i).limit(dim_bag);
            List<org.apache.spark.sql.Row> IDs;
            IDs = p.select("id").collectAsList();
            for(int j=0; j < IDs.size(); j++) {
                learn(PointToDoubleVec2(pset.where("id='"+IDs.get(j).getInt(0)+"'").first()));
            }
        }
    }else{
        List<org.apache.spark.sql.Row> IDs;
        IDs = pset.select("id").collectAsList();
        for(int i=0; i<epoch; i++)
            for(int j=0; j < IDs.size(); j++) {
                learn(PointToDoubleVec2(pset.where("id='"+IDs.get(j).getInt(0)+"'").first()));
            }
    }
    break;
```

Implementazione estrazione punti train BVQ.

Nota: in questa implementazione il numero di punti usati per la SOM e per il BVQ è identico, quindi le fasi di training impiegheranno lo stesso tempo. In realtà, il training SOM impiega n volte il tempo di bvq in quanto viene eseguito una volta per classe. Inoltre, all'aumentare dei dati aumentano i tempi delle fasi di split in training e test set e di sample dei punti in bag.

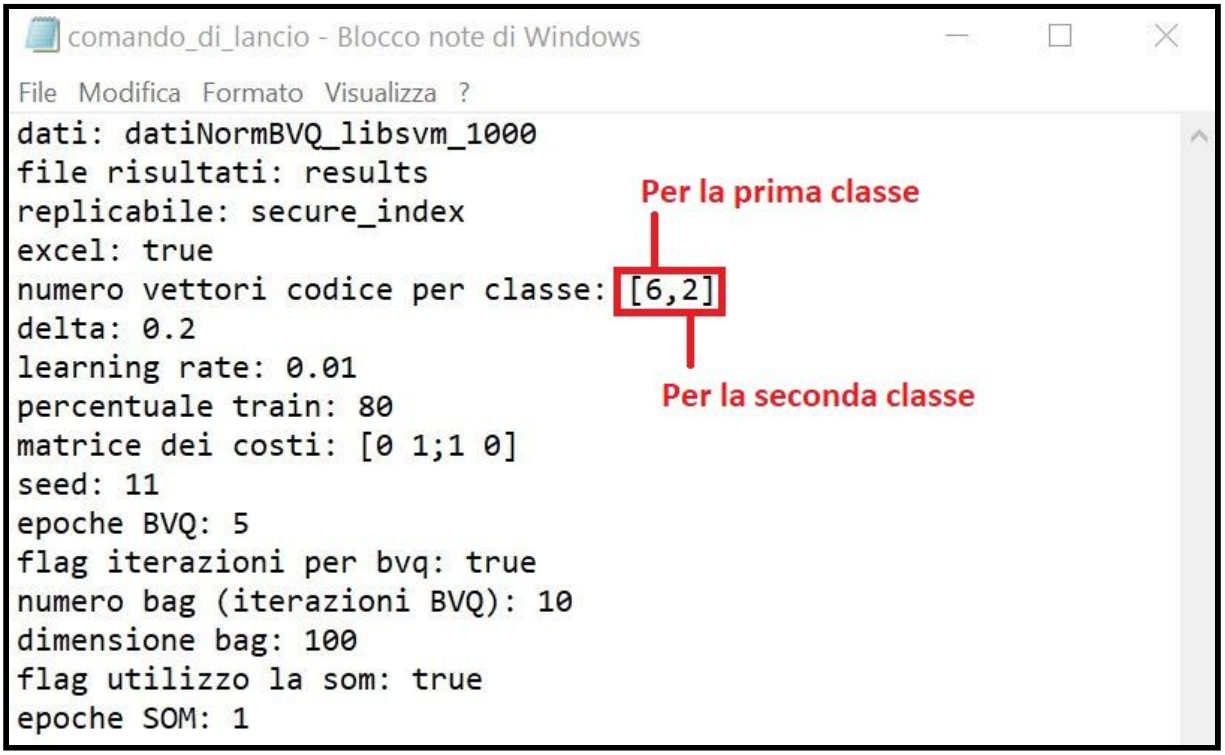
4. Ulteriori Miglioramenti

4.1 Garantire un miglior uso dei bag

Al fine di un ulteriore miglioramento della replicabilità e della coerenza dei dati presi per fare training, si è deciso di uniformare i dati che vengono presi dal SOM e dal BVQ. In sostanza, verranno usati gli stessi bag di punti per entrambi gli algoritmi. Questa operazione renderà anche i tempi di esecuzione di entrambe le procedure molto simili.
La modifica è effettiva non solo per la versione "secure_index", ma per tutte le versioni del codice, in modo da rendere anche più replicabili e coerenti gli esperimenti.

4.2 Cambiamento numero vettori codice

L'ultima feature che è stata aggiunta al programma è la seguente: è ora possibile impostare il numero di vettori codice per ciascuna delle due classi dal file contenente i comandi di lancio. Questa cosa, se si ha conoscenza pregressa sui punti presenti nel dataset e se sfruttata nel modo giusto può garantire una maggiore accuratezza.
Viene quindi mostrata di seguito l'immagine che indica come impostare il numero di vettori codice nel file di lancio.



5. Esperimenti

L'obiettivo che ci siamo posti in questa sezione è di testare le potenzialità e la scalabilità dell'algoritmo portando avanti una serie di esperimenti. In particolare, andremo a vedere le differenze che si hanno aumentando la quantità di dati data in pasto all'algoritmo e giocando con alcuni parametri.
Si testerà inoltre se effettivamente i punti vengono presi in modo deterministico o meno.

5.1 Test Ordine Punti Durante il Training

Al fine di verificare la ripetibilità del training si sono stampati a console i punti che venivano presi per fare training. Si è potuto così constatare che l'ordine di esecuzione è lo stesso.
Lo stesso tipo di test è stato fatto verificando che per due esecuzioni diverse, usando lo stesso seed, si ottenevano gli stessi vettori codice finali come risultato (a parità di numero di vettori codice impostato).
Questa cosa ovviamente non accade per la vecchia versione basata sugli RDD, la cui esecuzione risulta diversa ogni volta.

5.2 Test sulla Quantità di Dati

Qui siamo andati ad analizzare le metriche, come Accuratezza e Tempo di Esecuzione, per valutare l'algoritmo deterministico all'aumentare dei dati. Sono stati usati tre dataset, composti rispettivamente da 200, 2000 e 20000 punti, distribuiti in due classi e identificati da due features. Mantenendo costante il numero di punti che si userà per il training (num_bag*dim_bag) si dovrebbe riscontrare un aumento dell'accuratezza media all'aumentare della grandezza del training preso.

La modalità operativa che si è scelto di usare è di realizzare un ciclo for che riesegue il programma 10 volte per ciascun test e che media sia i tempi di esecuzione che l'accuratezza.
In console si otterrà un risultato simile al seguente.

Esempio valori medi per 10 esecuzioni del programma con seed diversi.

Nella seguente tabella sono riportati i risultati ottenuti dove "ACC" è l'accuratezza media ottenuta, "SOM" è il tempo medio dell'esecuzione del SOM, "BVQ" quello medio dell'esecuzione del BVQ stesso e "TOT" quello medio dell'esecuzione totale del programma.

Dataset→ **Training con dim_bag*num_bag punti** ↓	**200 punti**	**2000 punti**	**20000 punti**	**Altre info** ↓ **replicable = secure_index**
50 punti	ACC: 0.61726 SOM: 6,982 s BVQ: 1,770 s TOT: 9,419 s	ACC: 0.57090 SOM: 6,147 s BVQ: 1,611 s TOT: 8,640 s	ACC: 0.58756 SOM: 6,322 s BVQ: 1,771 s TOT: 9,695 s	**Numero Vettori Codice = [5,5]**
500 punti	ACC: 0.63083 SOM: 1,07 m BVQ: 22,146 s TOT: 1,15 m	ACC: 0.57109 SOM: 38,627 s BVQ: 14,32 s TOT: 55,824 s	ACC: 0.59439 SOM: 51,152 s BVQ: 16,681 s TOT: 1,02 m	
1000 punti	ACC: 0.64270 SOM: 1,28 m BVQ: 32,423 s TOT: 1,44 m	ACC: 0.57631 SOM: 1,38 m BVQ: 27,909 s TOT: 1,52 m	ACC: 0.59690 SOM: 1,38 m BVQ: 31,496 s TOT: 1,57 m	
50 punti	ACC: 0.54866 SOM: 6,965 s BVQ: 1,758 s TOT: 8,330 s	ACC: 0.53856 SOM: 6,882 s BVQ: 1,565 s TOT: 8,580 s:	ACC: 0.56864 SOM: 6,802 s BVQ: 1,718 s TOT: 8,564 s	**Numero Vettori Codice = [3,7]**
500 punti	ACC: 0.55207 SOM: 35,902 s BVQ: 15,708 s TOT: 51,43 s	ACC: 0.54172 SOM: 37,782 s BVQ: 13,409 s TOT: 53,728 s	ACC: 0.56935 SOM: 35,589 s BVQ: 15,431 s TOT: 55,693 s	
1000 punti	ACC: 0.56301 SOM: 1,27 m BVQ: 32,560 s TOT: 1,41 m	ACC: 0.53987 SOM: 1,25 m BVQ: 27,676 s TOT: 1,49 m	ACC: 0.56998 SOM: 1,21 m BVQ: 31,771 s TOT: 1,55 m	

Con questi esperimenti possiamo constatare alcune cose:

- A parità di dataset, aumentando il training si ottiene un'accuratezza migliore, come c'era da aspettarsi;
- A parità di training, aumentando il dataset c'è una diminuzione dell'accuratezza;
- La scelta del numero di vettori codice influisce pesantemente sui risultati ottenuti dall'algoritmo;
- Le dimensioni del dataset influenzano pesantemente le performance in termini di tempo delle fasi di training, mentre intaccano meno del previsto i tempi di split in training_set e test_set e di estrazione dei bag.

Gli esperimenti sono stati eseguiti su una macchina con 8GB di RAM e un processore Intel i5-8250U a 3.3 Ghz e dotato di SSD. Queste sono le caratteristiche hardware che più hanno influenzato i tempi dei vari test.

6. Conclusioni

Giunti al termine dello sviluppo di questa implementazione dell'algoritmo BVQ possiamo tirare le somme. La nuova versione risolve parecchi problemi della vecchia tra cui la ripetibilità dell'esperimento, l'uso delle ultime tecnologie messe a disposizione da Spark, la conversione al formato csv per importare i dati su cui lavora l'algoritmo, la possibilità di impostare il numero di vettori codici per classe e l'allineamento delle fasi di training della SOM e della BVQ in termini di estrazione di dati e di numero di punti utilizzati.

Per il futuro, è possibile ulteriormente migliorare l'algoritmo in modo da poter funzionare con più di due features e rimane la necessità di verificare operativamente in un contesto effettivamente parallelizzato (es: cluster reale) come si comporta l'algoritmo in tutte le sue forme, sia in termini di ripetibilità sia in termini di performance.