Report on Big Data project: 2015 Flight Delays

Giovanelli Joseph - Mat. 0000835168
Pisano Giuseppe - Mat. 0000841688

26 marzo 2020

Indice

1 Introduzione **3**
1.1 Descrizione del dataset . . . 3
1.1.1 Descrizione dei file . . . 3

2 Preparazione dei dati **5**
2.0.1 Data pre processing . . . 5

3 Jobs **7**
3.1 Airlines Job . . . 7
3.1.1 MapReduce . . . 7
3.1.2 Spark . . . 10
3.1.3 SparkSQL . . . 13
3.2 Airports Job . . . 16
3.2.1 MapReduce . . . 16
3.2.2 Spark . . . 18
3.2.3 SparkSQL . . . 20
3.2.4 Tableu . . . 23

4 Miscellaneous **25**
4.1 Machine Learning . . . 25
4.1.1 Pre processing . . . 26
4.1.2 Addestramento . . . 27
4.2 Struttura del progetto . . . 28

1 Introduzione

1.1 Descrizione del dataset

Il dataset "2015 Flight Delays and Cancellation" è composto da tre tabelle:

- **airlines**, contenente le informazioni riguardo alle compagnie aeree;
- **airports**, contenente le informazioni riguardo agli aeroporti degli USA;
- **flights**, contente le informazioni riguardo ai voli.

La tabella flights è la più corposa (192 MB) ed è anche quella con le informazioni più interessanti. Le altre due sono tabelle di servizio che aggiungono informazioni alla precedente. Il dataset è disponibile all'indirizzo
https://www.kaggle.com/usdot/flight-delays.

1.1.1 Descrizione dei file

Di seguito si offre una breve descrizione dei campi significativi utilizzati nei job.

airlines.csv Questo file contiene solo due colonne, ambedue utilizzate:

- iata_code, codice univoco della compagnia;
- airline, nome completo della compagnia.

airports.csv Questo file contiene sette colonne, delle quali quelle utilizzate sono:

- iata_code, codice univoco dell'aeroporto;
- airport, nome completo dell'aeroporto;
- latitude e longitude, rispettivamente, latitudine e longitudine dell'aeroporto (utilizzati unicamente per il machine learning);
- state, stato dell'aeroporto (utilizzato unicamente per il machine learning).

flights.csv Questo file contiene trentuno colonne, delle quali quelle utilizzate sono:

- month e day_of_week, mese e giorno della settimana in cui il volo è avvenuto (utilizzati unicamente per il machine learning);
- airline, iata_code della compagnia aerea;
- origin_airport e destination_airport, iata_code degli aeroporti di, rispettivamente, origine e destinazione (quest'ultimo utilizzato unicamente per il machine learning);
- scheduled_departure, orario di partenza programmato (in formato hhmm);

- taxi_out, tempo trascorso dalla partenza al gate al decollo effettivo (in minuti). Fornisce una stima del traffico in aeroporto;
- distance, distanza dall'aeroporto di origine a quello di destinazione (utilizzato unicamente per il machine learning);
- arrival_delay, ritardo complessivo del volo (in minuti);
- cancelled e diverted, valorizzati a 1 se il volo è stato, rispettivamente, cancellato o dirottato (utilizzati in fase di pre processing).

Per quanto riguarda quest'ultimo file (flights), le colonne che offrono informazioni sulla durata del volo, in caso di cancellazione o dirottamento, saranno nulle. Di seguito un esempio:

flights

AIRLINE	ORIGIN_AIRPORT	SCHEDULED_DEPARTURE	TAXI_OUT	ARRIVAL_DELAY	DIVERTED	CANCELLED
AS	ANC	135			0	1
DL	SLC	140	43	10	0	0

Figura 1: Esempio di dipendenze fra le colonne d'interesse del dataset flights

2 Preparazione dei dati

L'utente utilizzato per il progetto è **jgiovanelli** la cui macchina di riferimento è **isi-vclust3.csr.unibo.it**. Di seguito si descrive la struttura dell'HDFS:

- cartella **flights-dataset** contenente a sua volta:
  - la cartella **clean** contenente il dataset risultante dalla fase di data preparation. All'interno, per ogni tabella, è prevista una cartella contenente i file part-* generati dal job di pre processing;
  - la cartella **raw** contenente i tre file .csv descritti precedentemente;
- cartella **logs** contenente i log di esecuzione relativi alle implementazioni di riferimento dei diversi job;
- cartella **outputs** contenente i risultati dei job divisi per implementazione (map-reduce, spark, spark-sql).
- cartella **tableau** contenente il file AirporstJob.twbx, dove viene graficato in vari modi l'output dell'AirportsJob tramite tale software;

2.0.1 Data pre processing

Per quanto riguarda questa fase è stato implementato un job in Spark-Sql che carica i file dalla cartella flights-dataset/raw e, dopo averli puliti, salva il risultato in flights-dataset/clean. Questo job non esegue il pre processing richiesto per la componente di machine learning, la quale verrà argomentata nella relativa sezione.

Nello specifico:

- tutti i file vengono privati del loro header;
- sulla tabella **airport** viene eseguita una proiezione dei soli campi utilizzati nelle query (iata_code, airport)
- sulla tabella **flights**:
  - viene eseguita una proiezione dei soli campi utilizzati nelle query (airline, origin_airport, scheduled_departure, taxi_out, arrival_delay)
  - vengono eliminati i record corrispondenti ai voli cancellati o dirottati;
  - vengono eliminati i record che non presentano un identificatore iata_code dell'aeroporto valido.

Il comando per lanciare tale job è il seguente:
spark2-submit –class org.queue.bd.PreprocessingJob exam/spark-sql-1.0.jar

Esso prevede:

- inputs, **flights-dataset/raw/flights.csv**, **flights-dataset/raw/airlines.csv**, **flights-dataset/raw/airports.csv**;
- outputs, **flights-dataset/clean/flights**, **flights-dataset/clean/airlines**, **flights-dataset/clean/airports**.

3 Jobs

3.1 Airlines Job

Ordinare le compagnie in ordine decrescente in base all'arrival_delay medio.

3.1.1 MapReduce

Il comando per lanciare tale job è il seguente:
hadoop jar exam/map-reduce-1.0.jar org.queue.bd.airlinesjob.AirlinesJob

I link all'application history di YARN sono:

- Summarize:
  http://isi-vclust0.csr.unibo.it:19888/jobhistory/job/job_1552648127930_3786
- Join:
  http://isi-vclust0.csr.unibo.it:19888/jobhistory/job/job_1552648127930_3787
- Sort:
  http://isi-vclust0.csr.unibo.it:19888/jobhistory/job/job_1552648127930_3788

Esso prevede:

- inputs, **flights-dataset/clean/flights**, **flights-dataset/clean/airlines**;
- outputs, **outputs/map-reduce/airlines**.

È composto da tre sotto job:

1. **Summarize**
   - **Mapper**, legge dal file flights e produce record aventi come chiave lo iata_code della compagnia aerea e come valore un oggetto composto, RichSum, contenente (arrival_delay, 1);
   - **Combiner**, aggrega i valori riferiti allo stesso iata_code producendo (sum(arrival_delay), count);
   - **Reducer**, aggrega i valori riferiti allo stesso iata code producendo (iata_code, avg(arrival_delay)).
2. **Join**
   - **Mapper**, riceve come input il risultato di Summarize e produce (iata_code, RichAirline). Quest'ultimo è un oggetto composto utilizzato nel Reducer per comprendere da che mapper arriva il record. Esso contiene potenzialmente sia il nome della compagnia, sia la media dell'arrival_delay, ma in questo mapper viene valorizzata solo la media;

- **Mapper**, legge dal file airlines e produce record aventi come chiave lo iata_code della compagnia aerea e come valore il RichAirline sopra descritto. In questo mapper viene valorizzato solo il nome;
- **Reducer**, unifica i valori riferiti allo stesso iata code producendo (nome, avg(arrival_delay)).

3. **Sort**

   - **Mapper**, riceve come input il risultato della Join e produce (avg(arrival_delay), nome). Questo scambia semplicemente chiave e valore allo scopo di ordinare i risultati sulla media. È infatti intrinseco nel paradigma l'ordinamento totale sulla chiave se il numero di Reducer è pari a 1 (parziale altrimenti);
   - **Reducer**, produce in output (nome, avg(arrival_delay)). È creato un comparator apposito per ordinare in ordine decrescente secondo la chiave. In caso di stessa media, i record vengono ordinati in ordine alfabetico sul nome della compagnia.

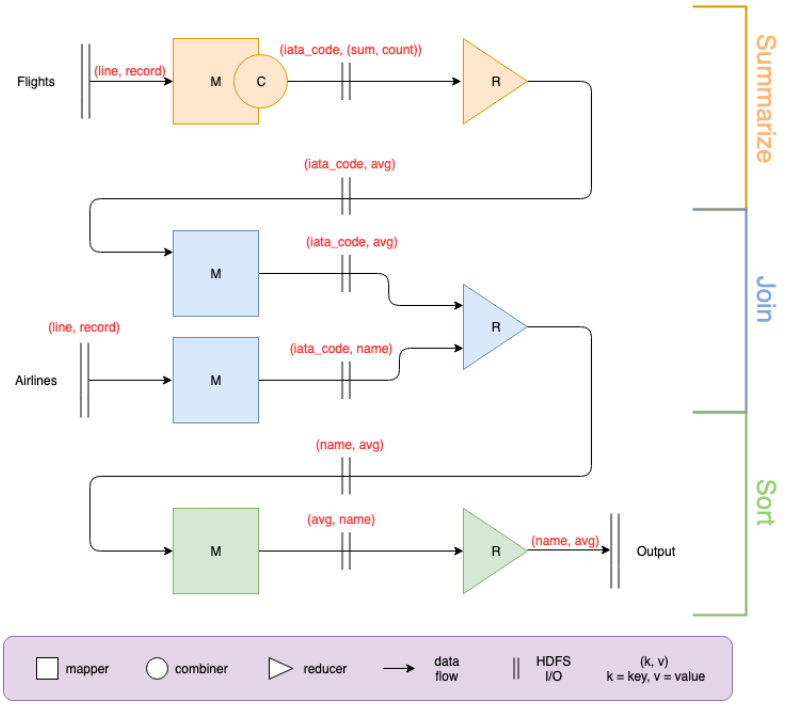

Figura 2: Schema Airlines Job (MapReduce)

Per quanto riguarda le considerazione sull'efficienza, per questo job sono stati valutati vari fattori:

- compressione dell'output del mapper;
- compressione dell'output del reducer;
- utilizzo di SequenceInputFormat e SequenceOutputFormat;
- numero di reducer;
- applicazione di combiner;
- aumento della data replication e quindi sfruttamento della data locality.

Di seguito si riportano i tempi di alcune delle prove effettuate. Queste comprendono solamente i risultati ottenuti con il job Summarize, poichè, essendo incaricato di processare una mole di dati maggiore, ha potuto beneficiare maggiormente delle ottimizzazioni.

compr.	comb.	n red.	repl	CPU	map	shuffle	merge	reduce	tot
		20	3	106	15	8	0	1	35
✓		20	3	100	14	7	0	1	34
	✓	20	3	88	14	7	0	1	32
	✓	1	3	48	14	5	0	0	27
✓	✓	1	3	47	14	4	0	0	27
✓	✓	1	5	47	14	4	0	0	26

Come si può osservare:

- con una mole di dati considerevole la compressione porta dei miglioramenti, soprattuto in fase di shuffling;
- stessa considerazione si può fare sull'utilizzo del combiner;
- il numero di reducer è l'ottimizzazione che porta più giovamento;

La configurazione finale prescelta per il Job Summarize è quella mostrata nell'ultima riga della tabella. Si specifica che le prove eseguite prevedono come input di questo job la tabella flights suddivisa in cinque porzioni, a seguito della fase di pre processing. Senza questo accorgimento sarebbe stato istanziato un solo mapper, data la grandezza in input inferiore a quella di un blocco HDFS. In tal modo invece i mapper utilizzati sono cinque. Da questo derivano le prove fatte sull'aumento della data replication. Infatti avendo un solo mapper queste non avrebbero avuto nessun impatto.

Per quanto concerne i tempi di Join e Sort, le tecniche di compressione non hanno portato a nessun miglioramento, anzi, hanno causato un leggero peggioramento delle performance. Tali accorgimenti sono stati attuati per ridurre i tempi di trasporto tra Mapper-Reducer e viceversa. Si ipotizza che il mancato miglioramento derivi dal ristretto volume dei dati, il cui trasporto non riesce

a compensare l'overhead computazionale dovuto dalla compressione stessa. Si è deciso quindi di non utilizzarli. La diminuzione del numero dei reducer si è dimostrata vincente anche nel caso del job Join. Il valore prescelto nella configurazione finale è 1.

3.1.2 Spark

Il comando per lanciare tale job è il seguente:
spark2-submit –num-executors 9 –class org.queue.bd.AirlinesJob exam/spark-1.0.jar

Esso prevede:

- inputs, **flights-dataset/clean/flights**, **flights-dataset/clean/airlines**;
- outputs, **outputs/spark/airlines**.

Il comportamento del job prevede:

- **lettura** della tabella flights dal file system;
- **selezione** delle colonne di interesse da flights;
- **aggregazione** dei voli in base alla compagnia aerea, con media sul ritardo di arrivo. È stata utilizzata la funzione aggregateByKey allo scopo di ridurre lo shuffling;
- **lettura** della tabella airlines dal filesystem;
- **join** tra le tabelle flights ed airlines per integrare il nome completo della compagnia alla media già calcolata;
- **ordinamento** delle compagnie in ordine decrescente secondo la media.

Per quanto concerne l'ottimizzazione sono state testate quattro soluzioni distinte. Nella figura 3 viene mostrato il DAG dell'implementazione base sopra descritta. Come si può notare viene effettuato uno shuffling dei dati, sia in fase di aggregazione, sia in quella di join. Si è cercato quindi di evitare questo comportamento forzando il partizionamento.

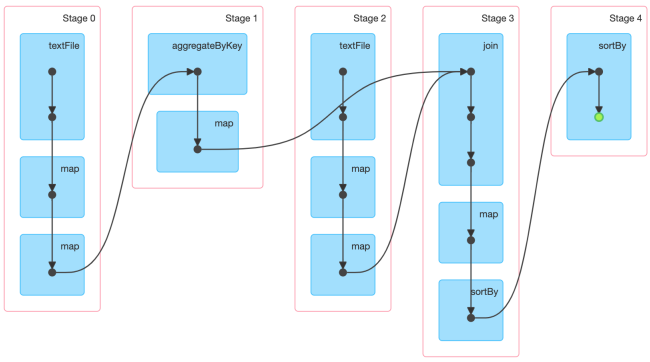

Figura 3: AirlinesJob - DAG senza partizionamento

Il risultato è mostrato nel diagramma 4, dove applicando un HashPartitioner sulla chiave di flights, nonchè attributo di aggregazione e condizione di join, si può notare come le operazioni vengano eseguita in pipeline. Tuttavia si è osservato che, pur avendo ridotto il numero di shuffling, il volume dei dati scambiati per il partizionamento è aumentato, dato il mancato sfruttamento dell'ottimizzazione data dall'aggregateByKey.

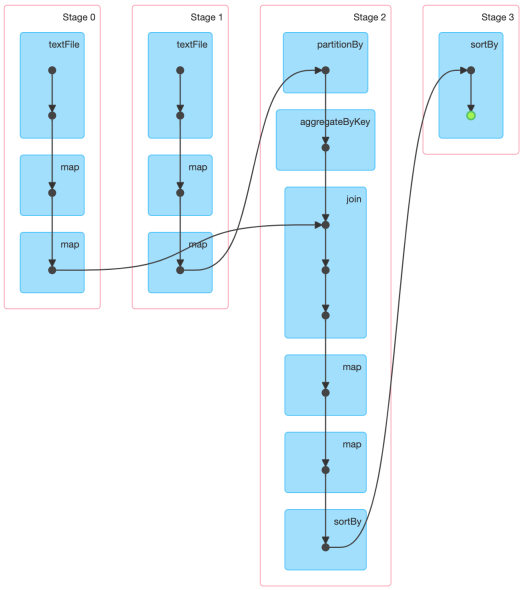

Figura 4: AirlinesJob - DAG con partizionamento prima dell'aggregateByKey

Si è arrivati quindi alla terza soluzione, figura 5, nella quale si cercano di conciliare i vantaggi delle versioni precedenti. Applicando il Partitioner dopo

l'aggregazione si aumenta il numero degli shuffling, ma si riduce la quantità di dati scambiata.

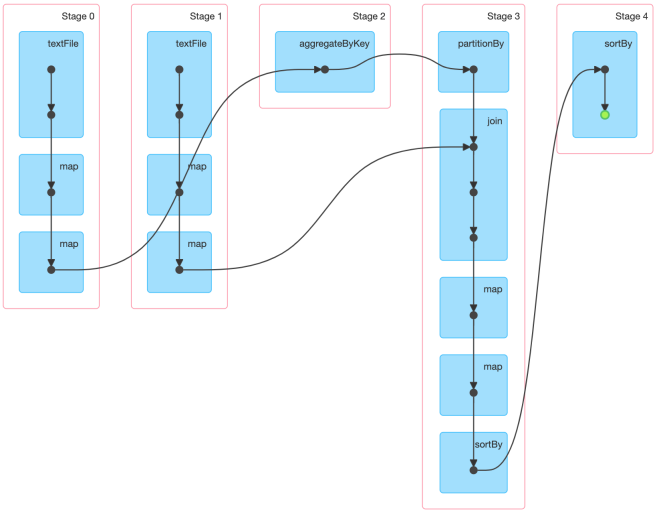

Figura 5: AirlinesJob - DAG con partizionamento dopo l'aggregateByKey

Quarta e ultima soluzione testata è quella basata sulle Broadcast Variables, figura 6. Come si può vedere dal DAG la computazione avviene in un unico stage, riducendo lo shuffling al minimo. Nello specifico è la tabella airlines ad essere condivisa tra i nodi del cluster.

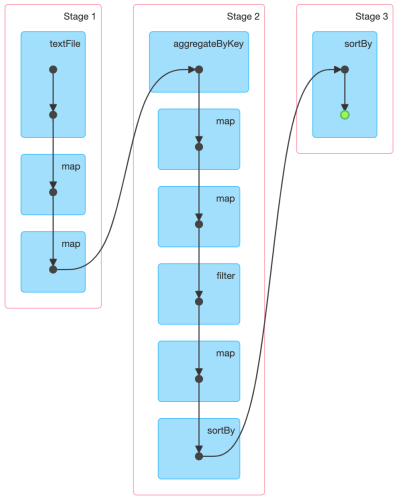

Figura 6: AirlinesJob - DAG con Broadcast Variables

Si è cercato inoltre di aumentare la parallelizzazione del lavoro incrementando il numero di executor. Per le prime tre soluzioni l'ottimo si è raggiunto con un numero pari a nove, mentre per quella basata sulle BV l'aumento di executor ha comportato solamente un incremento del numero di nodi a cui far reperire la tabella airlines, con un conseguente peggioramento nelle prestazioni.

La soluzione tre e la soluzione quattro, al loro ottimo, si sono dimostrate equivalenti. Non sono comunque riuscite ad eguagliare i risultati raggiunti dalla soluzione base, che con un numero di executor pari a nove, ha completato il lavoro nel minor tempo.

Per avere i dettagli sulle soluzioni citate si rimanda alla history YARN. Di seguito i riferimenti:

- Soluzione 1, base
  http://isi-vclust0.csr.unibo.it:18089/history/application_1552648127930_3792/jobs/
- Soluzione 2, con partizionamento precedente all'aggregazione
  http://isi-vclust0.csr.unibo.it:18089/history/application_1552648127930_3656/jobs/
- Soluzione 3, con partizionamento successivo all'aggregazione
  http://isi-vclust0.csr.unibo.it:18089/history/application_1552648127930_3580/jobs/
- Soluzione 4, basata sulle Broadcast Variables
  http://isi-vclust0.csr.unibo.it:18089/history/application_1552648127930_3593/jobs/

3.1.3 SparkSQL

Il comando per lanciare tale job è il seguente:
spark2-submit –num-executors 9 –class org.queue.bd.AirlinesJob exam/spark-sql-1.0.jar

Esso prevede:

- inputs, **flights-dataset/clean/flights**, **flights-dataset/clean/airlines**;
- outputs, **outputs/spark-sql/airlines**.

Il link all'application history di YARN è:

- **http://isi-vclust0.csr.unibo.it:18089/history/application_1552648127930_3794**

Il contenuto del job è espresso chiaramente dal piano di esecuzione logico generato:

```
== Analyzed Logical Plan ==
AIRLINE: string, AVERAGE_DELAY: double
Sort [average_delay#36 DESC NULLS LAST], false
+- Repartition 1, false
   +- Project [AIRLINE#3, AVERAGE_DELAY#36]
      +- Join Inner, (AIRLINE#12 = IATA_CODE#2)
         :- SubqueryAlias SF
         :  +- Aggregate [AIRLINE#12], [AIRLINE#12,
               avg(ARRIVAL_DELAY#26) AS AVERAGE_DELAY#36]
         :     +- SubqueryAlias flights
         :        +- Project [AIRLINE#12, ARRIVAL_DELAY#26]
         :           +- Project [AIRLINE#12, ARRIVAL_DELAY_TEMP#16,
                        UDF(ARRIVAL_DELAY_TEMP#16) AS ARRIVAL_DELAY#26]
         :              +- Project [AIRLINE#12, ARRIVAL_DELAY_TEMP#16]
         :                 +- LogicalRDD [AIRLINE#12, ORIGIN_AIRPORT#13,
                           SCHEDULED_DEPARTURE#14, TAXI_OUT#15,
                           ARRIVAL_DELAY_TEMP#16]
         +- SubqueryAlias A
            +- SubqueryAlias airlines
               +- LogicalRDD [IATA_CODE#2, AIRLINE#3]
```

Questo è generato dalla query:

```
select A.AIRLINE, SF.AVERAGE_DELAY
from (select AIRLINE, avg(ARRIVAL_DELAY) as AVERAGE_DELAY
      from flights
      group by AIRLINE) SF
join airlines A on SF.AIRLINE = A.IATA_CODE
order by SF.AVERAGE_DELAY desc
```

L'ottimizzatore Catalyst ha provveduto in automatico alla creazione di un piano di esecuzione ottimizzato, come si può notare dallo schema 7. Per questo motivo, tutte le prove fatte hanno portato a dei risultati peggiori di quelli base. L'unico accorgimento di un qualche valore è stato quello di portare il numero di executor a nove. Per questo job sono state utilizzate le API che permettono di dichiarare l'elaborazione da eseguire con sintassi SQL tramite stringa.

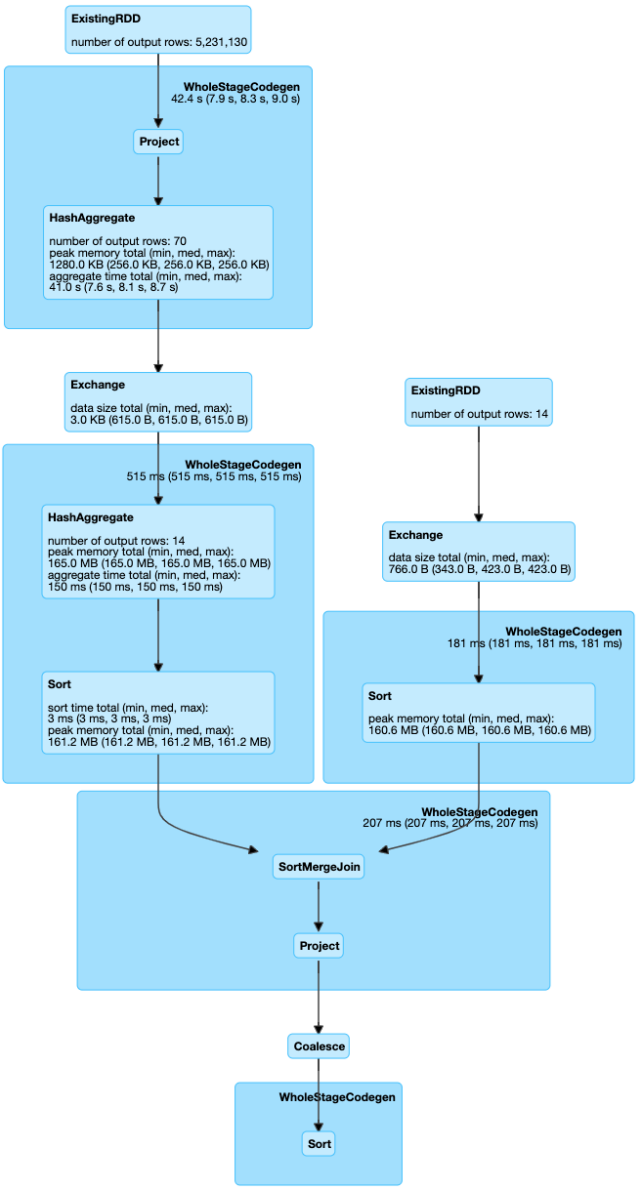

Figura 7: Si può notare come il processo di aggregazione sia ottimizzato dal punto di vista dello shuffling dei dati. Infatti, prima della redistribuzione di flights, viene eseguita un'aggregazione locale che porta a una notevole diminuzione nel volume dei record. Parallelamente al calcolo della media viene eseguito il processamento della tabella Airlines, unita a quella Flights tramite un'operazione di SortMergeJoin. Per concludere viene eseguito lo shuffling dei dati su un'unica partizione in modo da procedere al loro ordinamento totale.

3.2 Airports Job

Per ogni aeroporto indicare il ritardo taxi_out medio, diviso per delle fasce orarie create. È previsto il plot dei risultati con tableau.

3.2.1 MapReduce

Il comando per lanciare tale job è il seguente:
hadoop jar exam/map-reduce-1.0.jar org.queue.bd.airportsjob.AirportsJob

I link all'application history di YARN sono:

- Summarize:
  http://isi-vclust0.csr.unibo.it:19888/jobhistory/job/job_1552648127930_3789
- Join:
  http://isi-vclust0.csr.unibo.it:19888/jobhistory/job/job_1552648127930_3790
- Sort:
  http://isi-vclust0.csr.unibo.it:19888/jobhistory/job/job_1552648127930_3791

Esso prevede:

- inputs, **flights-dataset/clean/flights**, **flights-dataset/clean/airports**;
- outputs, **outputs/map-reduce/airports**.

Di seguito si fornisce e commenta lo schema del job.

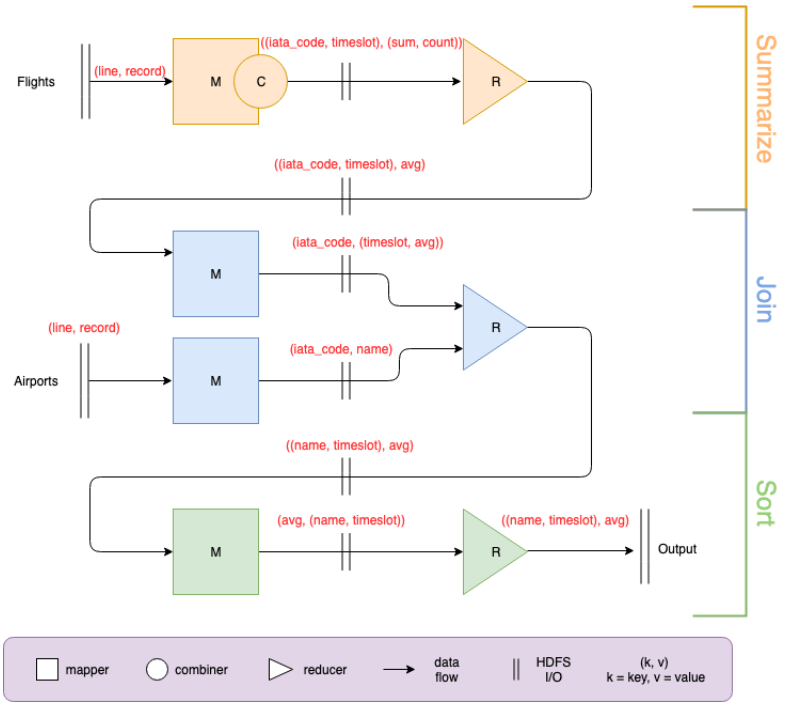

Figura 8: Schema Airports Job (MapReduce)

La struttura è del tutto identica a quella già illustrata nell'Airline Job. Per tale motivo di seguito verranno argomentate solamente le differenze con il Job precedente.

1. **Summarize**, in questo caso lo iata_code è riferito all'aeroporto di origine e sum e count fanno riferimento al taxi_out. Inoltre l'aggregazione non viene effettuata su una key semplice, ma su un oggetto composto, RichKey, il quale contiene iata_code e timeslot. Quest'ultimo viene ricavato tramite un'apposita funzione applicata allo scheduled_departure del volo in oggetto;

2. **Join**, unica differenza rispetto all'Airlines Job sta nell'aver effettuato la join con le informazioni provenienti dal file airports, sempre però con la finalità di ottenere il nome completo dell'aeroporto, non presente nelle informazioni di flights. Sono state necessarie delle modifiche in modo che il contenuto informativo del value (RichAirport in questo caso) permettesse la gestione del timeslot, ma nel complesso la logica rimane la medesima;

3. **Sort**, per questa fase è stato utilizzato lo stesso Job visto per l'ordinamento delle compagnie aeree. La differenza fra le key derivanti dalle fasi di Join dei due Job è stata astratta utilizzando un valore testuale che ne rappresentasse il contenuto informativo in modo trasparente a questo job

di Sort. Tale job si può quindi limitare a sccambiare key e value, effettuare il sort, e quindi riscambiare gli elementi della tupla.

Per quanto riguarda le considerazioni sull'efficienza, per questo job sono stati valutati gli stessi fattori visti sopra. Anche in questo caso si riportano i soli risultati relativi al job di Summarize.

compr.	comb.	n red.	repl	CPU	map	shuffle	merge	reduce	tot
		20	3	157	18	7	1	1	40
✓		20	3	148	18	7	1	1	37
	✓	20	3	110	19	7	0	0	39
	✓	1	3	93	23	5	0	0	39
✓	✓	1	3	95	23	5	0	0	39
✓	✓	1	5	91	22	4	0	0	36

Per quanto riguarda la configurazione definitiva del Summarize, questa, come previsto, vista la stessa natura dei due Job, rimane invariata rispetto al caso precedente. Sempre analoghe le configurazioni e considerazioni sugli altri due Job (Sort e Join).

3.2.2 Spark

Il comando per lanciare tale job è il seguente:
spark2-submit –class org.queue.bd.AirportsJob exam/spark-1.0.jar

Esso prevede:

- inputs, **flights-dataset/clean/flights**, **flights-dataset/clean/airports**;
- outputs, **outputs/spark/airports**.

Il comportamento del job prevede:

- **lettura** della tabella flights dal file system;
- **selezione** delle colonne di interesse da flights;
- **aggregazione** dei voli in base all'aeroporto di origine e al timeslot dell'ora di partenza prevista. Questo viene ricavato, come nell'implementazione map-reduce, dal campo scheduled_departure. Viene in questo modo prodotta la media del ritardo taxi_out. Anche in questo caso è stata utilizzata la funzione aggregateByKey allo scopo di ridurre lo shuffling;
- **lettura** della tabella airports dal file system;
- **join** tra le tabelle flights ed airports per integrare il nome completo dell'aeroporto di origine alla media già calcolata.

Come si può notare la struttura è analoga a quella dell'AirlinesJob. L'unica differenza consiste nell'utilizzo di una chiave composta in fase di aggregazione, differente da quella valutata in fase di join.

Per ottenere dei risultati comparabili all'AirlinesJob si rende necessario un mapping di airports, in seguito al quale, integrando il timeslot nella chiave, ogni record viene ripetuto per ogni suo possibile valore (MORNING, AFTERNOON, EVENING. NIGHT). In questo modo, però, si quadruplicano i valori di tale tabella, rendendo la soluzione non scalabile all'aumentare del numero di record.

Ulteriore approccio vede l'utilizzo di un Custom Partitioner in modo da forzare il partizionamento solo su una parte della chiave di aggregazione, iata_code dell'aeroporto di origione, cercando così di evitare lo shuffling, sia in fase di aggregazione che di join.

Il comportamento desiderato non si è però verificato in quanto, dovendo ri-mappare la chiave dopo l'aggregazione per uniformarla al join, Spark effettua una ridistribuzione dei dati. Inoltre in tale soluzione non viene sfruttata l'aggregazione locale data dall'aggregateByKey, andando così ad aumentare la quantità dei dati in shuffling. Il DAG di riferimento è mostrato in figura 9

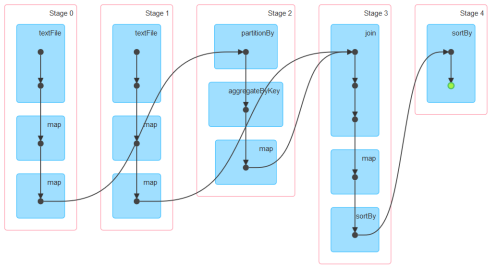

Figura 9: AirportsJob - DAG con partizionamento custom prima dell'aggregazione

Si è quindi tentato un partizionamento successivo all'aggregazione e al mapping, in modo da ridurre lo shuffling in fase di join. Il DAG di tale soluzione è mostrato in figura 10.

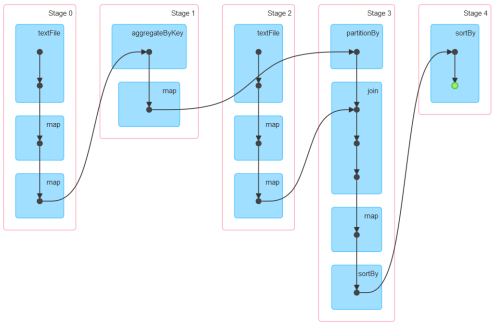

Figura 10: AirportsJob - DAG con partizionamento successivo all'aggregazione

Come per Airlines si sono fatte delle prove con un numero di executor maggiore a quello di default, riconfermando il nove come parametro ottimale.

A differenza dell'Airlines Job, però, è emersa come soluzione migliore quella basata sulle Broadcast Variables, a pari merito con quella con partizionamento successivo all'aggregazione. La prima è stata adottata come implementazione di riferimento. La soluzione base ha invece prodotto dei risultati leggermente inferiori. Sia di questa, che della versione con BV, non si fornisce il DAG in quanto il comportamento rimane analogo a quello di Airlines.

Di seguito si riportano i riferimenti alla history YARN relativi alle prove sopra descritte, dove si può effettivamente osservare la differenza nei tempi di esecuzione e nella quantità di dati scambiati nelle diverse soluzioni.

- Soluzione 1, base
  http://isi-vclust0.csr.unibo.it:18089/history/application_1552648127930_3658/jobs/
- Soluzione 2, con partizionamento precedente all'aggregazione
  http://isi-vclust0.csr.unibo.it:18089/history/application_1552648127930_3646/jobs/
- Soluzione 3, con partizionamento successivo all'aggregazione
  http://isi-vclust0.csr.unibo.it:18089/history/application_1552648127930_3657/jobs/
- Soluzione 4, basata sulle Broadcast Variables
  http://isi-vclust0.csr.unibo.it:18089/history/application_1552648127930_3793/jobs/

3.2.3 SparkSQL

Il comando per lanciare tale job è il seguente:
spark2-submit –num-executors 9 –class org.queue.bd.AirportsJob exam/spark-sql-1.0.jar

Esso prevede:

- inputs, **flights-dataset/clean/flights**, **flights-dataset/clean/airports**;
- outputs, **outputs/spark-sql/airports**.

Il link all'application history di YARN è:

- **http://isi-vclust0.csr.unibo.it:18089/history/application_1552648127930_3795**

In questo caso il piano di esecuzione logico generato risulta essere meno chiaro del precedente, a causa della differente modalità di input dati utilizzata (per tale motivo quello mostrato qui è stato leggermente modificato in modo da aumentarne la leggibilità), ma rimane comunque abbastanza esauriente:

```
== Analyzed Logical Plan ==
AIRPORT: string, TIME_SLOT: string, avg(TAXI_OUT): double
Sort [avg(TAXI_OUT)#26 DESC NULLS LAST], false
+- Repartition 1, true
   +- Project [AIRPORT#4, TIME_SLOT#14, avg(TAXI_OUT)#26]
      +- Join Inner, (ORIGIN_AIRPORT#13 = IATA_CODE#3)
         :- Aggregate [ORIGIN_AIRPORT#13, TIME_SLOT#14], [ORIGIN_AIRPORT#13,
            TIME_SLOT#14, avg(TAXI_OUT#15) AS avg(TAXI_OUT)#26]
         :  +- SerializeFromObject[YAFlight]
         :     +- ExternalRDD [obj#12]
         +- SerializeFromObject [YAAirport]
            +- ExternalRDD [obj#2]
```

Questo è generato dalla query:

```
select A.AIRPORT, SF.AVERAGE_TAXI_OUT
from (select AIRPORT, IF(SCHEDULED_DEPARTURE<0600, 'NIGHT',
                         IF(SCHEDULED_DEPARTURE<1200, 'MORNING',
                         IF(SCHEDULED_DEPARTURE<1800, 'AFTERNOON',
                         'EVENING'))) AS TIME_SLOT,
                         avg(TAXI_OUT) as AVERAGE_TAXI_OUT
      from airports
      group by AIRPORT, TIME_SLOT) SF
join airports A on SF.AIRPORT = A.IATA_CODE
order by SF.AVERAGE_TAXI_OUT desc
```

Anche in questo caso l'ottimizzatore ha provveduto in automatico alla creazione di un ottimo piano di esecuzione, difficilmente ottimizzabile.

Per questo job sono state utilizzate le API fluenti, e non la sintassi SQL.

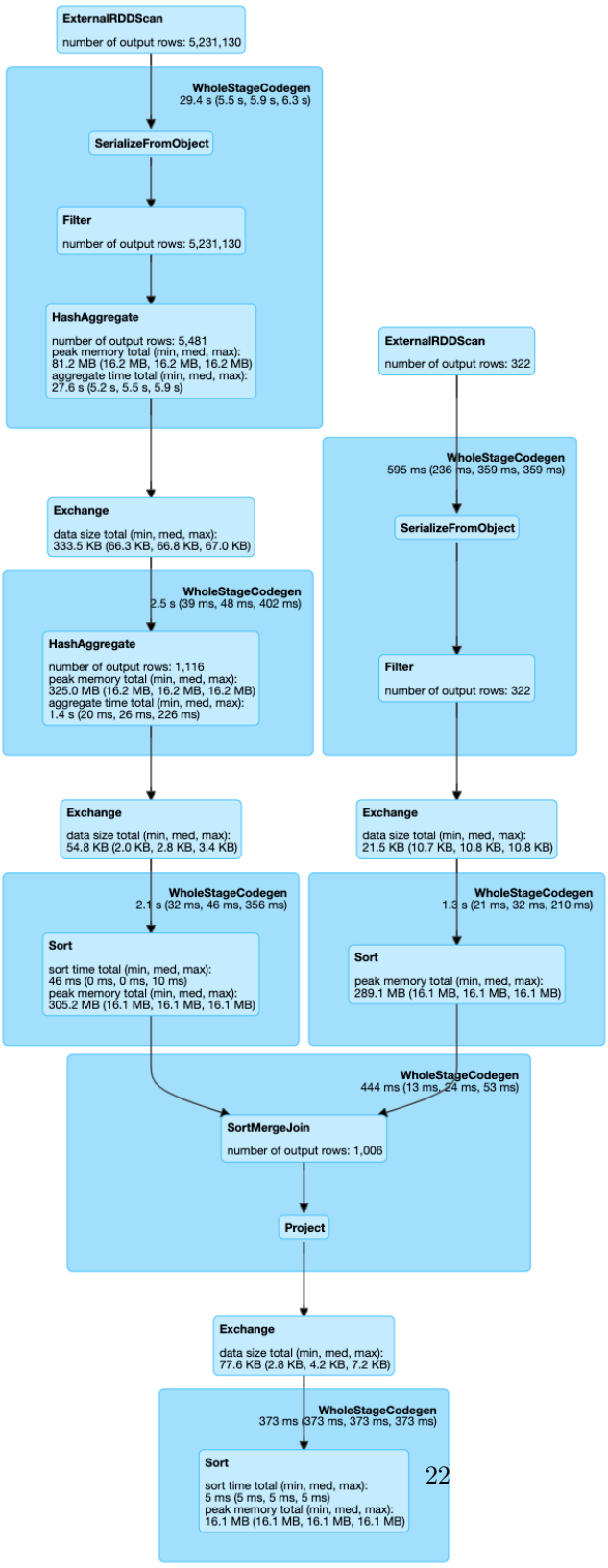

Figura 11: Il piano prodotto per questo Job risulta del tutto simile a quello generato per Airlines, rendendo valide anche in questo caso tutte le considerazioni fatte sopra.

3.2.4 Tableu

I risultati prodotti dal job sono stati importati all'interno del software Tableau e utilizzati per graficare e quindi analizzare gli output. Come formato di input è stato utilizzato il csv. Infatti, entrambi i job, in tutte e tre le implementazioni, sono stati configurati in modo tale da salvare i risultati in questo formato.

Sotto vengono riportati due grafici generati con tale software. Il primo, un bar chart dove le dimensioni considerate sono l'aeroporto e il Timeslot e la misura in esame è il ritardo taxi out, mostra gli aeroporti ordinati sulla somma di questa nell'arco della giornata (figura 12).

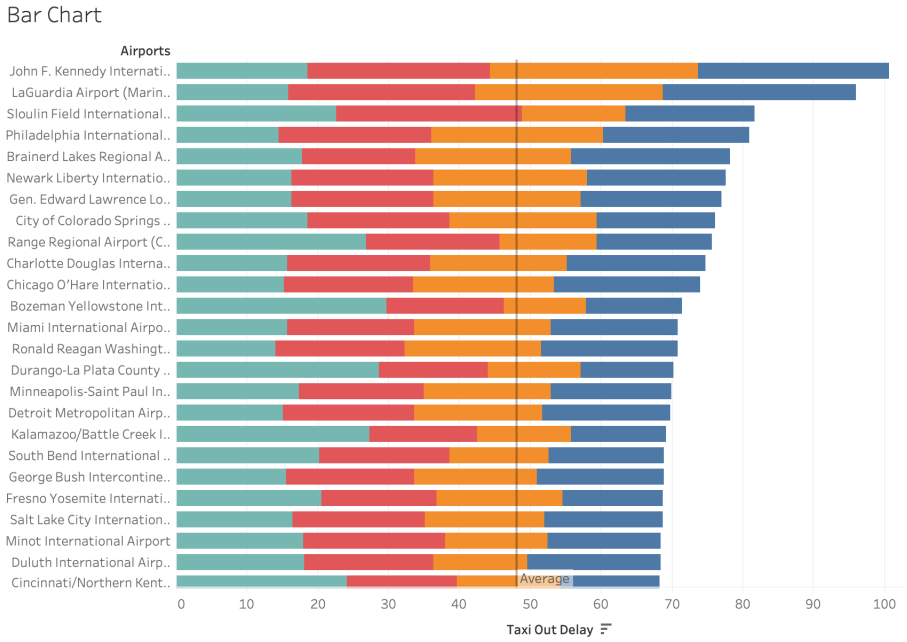

Figura 12: Bar chart che vede Airoports e Time Slot come dimensioni e Taxi Out Delay come misura

Il secondo grafico mostra invece quattro box plot, uno per ogni TimeSlot (figura 13). Questo è stato fatto per scoprire come i taxi out delay degli areoporti considerati variassero in base al TimeSlot. Si è verificato come le distanze tra ciascun quartile e la rispettiva mediana siano molto simili nei quattro box plot, denotando quindi la simmetria delle distribuzioni. Inoltre le distanze interquartili variano di poco fra loro, il che ci porta a dire che il grado di dispersione delle distribuzioni sia pressoché lo stesso nei quattro casi. Sono presenti inoltre molti outliers, tutti nella parte alta del grafico.

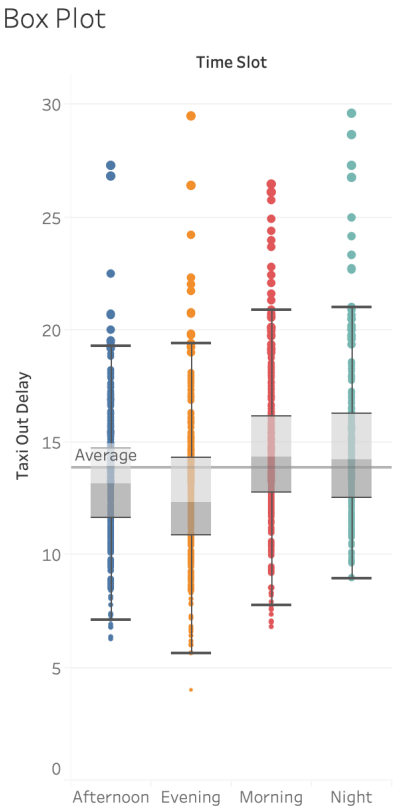

Figura 13: Box Plot che mostrano la distrubuzioni degli aeroporti sul ritardo medio, secondo le diverse fascie orarie

Sono stati inoltre incrociati i risultati riscontrati nei due grafici. Si sono presi i primi elementi del bar chart e li si è andati a cercare nei box plot, scoprendo che questi presentano come outliers per lo più sempre gli stessi elementi. Infatti, un outlier presente su un boxplot ha molta probabilità di essere outlier negli altri boxplot, e quindi di avere una somma di taxi out delay alta, presentandosi tra le prime posizioni del bar chart.

4 Miscellaneous

4.1 Machine Learning

È stato utilizzato il data set flights delays anche per effettuare delle prove in merito alle funzionalità di Machine Learning offerte da SparkML. Nello specifico si è tentato di inferire dalle caratteristiche del volo se questo potesse essere soggetto a ritardo. Sono stati allenati due modelli differenti, di cui si forniscono i riferimenti YARN con relativi comandi per l'esecuzione:

- Decision Tree,
  - **http://isi-vclust0.csr.unibo.it:18089/history/application_1552648127930_3831/jobs**
  - **spark2-submit –num-executors 9 –class org.queue.bd.ml.MachineLearningJob exam/spark-sql-1.0.jar tree**
- Logistic Regression,
  - **http://isi-vclust0.csr.unibo.it:18089/history/application_1552648127930_3832/jobs**
  - **spark2-submit –num-executors 9 –class org.queue.bd.ml. exam/spark-sql-1.0.jar logistic**

Essi prevedono:

- inputs, **outputs/spark-sql/machine-learning/ml-preprocessing/**
- outputs, **outputs/spark-sql/machine-learning/ml-results/**

Sono stati creati ulteriori due job di supporto: l'MLSamplingJob e l'MLPreprocessingJob. Il secondo è incaricato di pulire i dati grezzi in modo da prepararli al processo di apprendimento automatico. Il primo, invece, dati gli scarsi risultati ottenuti tramite i due modelli, è stato usato per campionare in modo casuale il data set. In questo modo si è potuto importare la sotto porzione dei dati prodotta (20%) all'interno di tool alternativi, come Weka. Così facendo è stato possibile, sia visualizzare i dati, che testare gli stessi modelli, verificando che il mancato raggiungimento del risultato non fosse dovuto ad un utilizzo scorretto di Spark MLlib.

Si è notato infatti come i dati fossero, per il problema preso in oggetto, molto ostici da classificare. Di seguito si offrono dei diagrammi dove in blu sono rappresentate le istanze di classe negativa (senza ritardo), in rosso quelle positive (con ritardo).

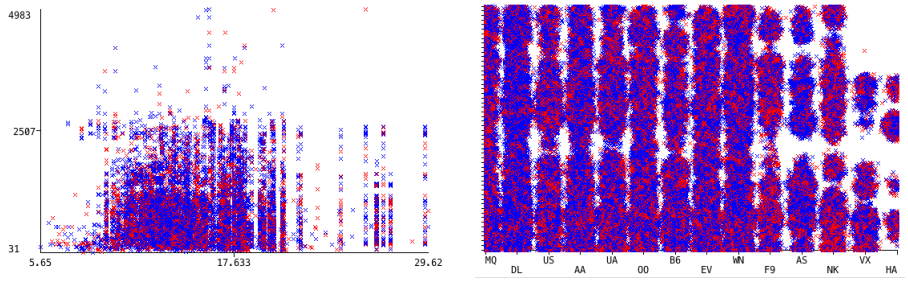

Figura 14: A sinistra sulle ascisse l'average taxi_out dell'aeroporto di origine e sulle ordinate la distanza coperta dal volo. A destra sulle ascisse la compagnia aerea e sulle ordinate lo stato dell'aeroporto di origine.

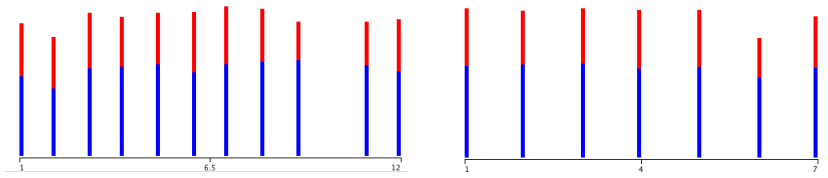

Figura 15: A sinistra la distribuzione delle classi in relazione al mese in cui il volo è avvenuto. A destra in relazione al giorno della settimana.

Ora si entrerà nello specifico del lavoro svolto all'interno del job di pre processing e di quello per l'addestramento vero e proprio.

4.1.1 Pre processing

Oltre ad applicare la pulizia del job di pre processing argomentato ad inizio relazione, si sono effettuati vari join, sia con le tabelle originali (airlines ed airports), che con i risultati dei job da noi realizzati (AirlinesJob ed AirportsJob). Il risultato di tale fase è una tabella contenente i seguenti campi:

- Airline, iata_code della compagnia aerea;
- TimeSlot, prodotto, come già visto in precedenza, da scheduled_departure;
- Month, mese in cui è avvenuto il volo;
- DayOfWeek, giorno della settimana in cui è avvenuto il volo;
- Distance, distanza compresa tra l'aeroporto di origine e quello di destinazione;
- OriginLatitudeArea e OriginLongitudeArea, fasce geografiche prodotte utilizzando delle udf a partire dalle coordinate reali dell'areoporto d'origine;

- OriginState;
- DestinationLatitudeArea e DestinationLongitudeArea, come sopra ma per l'aeroporto di destinazione;
- DestinationState;
- OriginAirport, iata_code dell'aeroporto;
- AverageAirlineDelay, ritardo d'arrivo medio delle compagnia aerea;
- AverageTaxiOut, taxi out medio nell'aeroporto d'origine;
- Delay, label della classe da predire estratta tramite una user defined function, 1 se il volo è in ritardo, 0 altrimenti.

Come si può notare, utilizzando le informazioni provenienti dai nostri job si è introdotta una distorsione del risultato. Infatti i campi AverageAirlineDelay e AverageTaxiOut portano con sé informazioni aggregate su tutti i voli, e non solo del sotto insieme di train. Si è notato come tali indici siano molto discriminanti per la risoluzione di questo problema, e come quindi apportino un particolare giovamento al risultato. Non è stato, però, previsto un meccanismo che calcolasse tali indici solo sul set di train, dato che la realizzazione di questo job non era programmata all'inizio di tale progetto.

4.1.2 Addestramento

Per rendere tale sezione il più snella possibile si procede ora ad elencare le fasi della Pipeline attuata:

- conversione degli attributi categorici descrittivi in indici (Airline, Timeslot, OriginState, DestinationState, OriginAirport, Delay);
- discretizzazione dei valori continui (Distance, AverageArrivalDelay, AverageTaxiOut);
- in caso di Logistic Regression si sono inoltre normalizzate le scale degli attributi numerici;
- divisione del dataset in Train (70%) e Test (30%) Set;
- addestramento del modello sulla base del training set;
- valutazione dei risultati su train e test set.

Il modello è riuscito a raggiungere solo il 64% di accuratezza nel caso di Decision Tree e 62% in quello di Regression. Osservando inoltre Precision, Recall e l'area sottostante la curva di ROC si può notare come il classificatore non riesca a distinguire in maniera chiara le istanze delle due classi, nonostante si siano attuate tecniche di pre processing dei dati coerenti con i modelli allenati. Si ritiene quindi, a fronte delle tecniche di data visualization utilizzate per

comprendere più a fondo il dominio, che tali classificatori non siano abbastanza espressivi da valutare un data set così complesso. Si è valutato l'utilizzo di un classificatore Bayesiano, data la sua buona capacità di predire in ambienti stocastici, ma il suo utilizzo avrebbe comportato una normalizzazione diversa da quella utilizzata, essendo presenti dei valori negativi che vanno in contrasto con le probabilità alla base di tale modello. Futuri sviluppi potrebbero prevedere l'utilizzo di classificatori più espressivi (Nearest Neighbor) o di tecniche di bagging.

4.2 Struttura del progetto

Per una navigazione più immediata del codice si procede a descriverne la struttura. Si è deciso di creare un multi-progetto e gestire le dipendenze tra i vari moduli tramite il build tool Gradle.

- modulo **common**, contente le dipendenze comuni tra le varie implementazioni
  - package **pojos**, classi di servizio per modellare le entità del dominio
  - package **utils**, classi di utlità comuni
- modulo **map-reduce**, contenente i job in tale implementazione
  - package **airlinesjob**, job riguardate le compagnie aeree
  - package **airportsjob**, job riguardante gli aereoporti
  - package **commons**, SortJob in comune a entrambi i job sopracitati
  - package **richobjects**, classi custom utilizzate come chiave e/o valore in MapReduce
- modulo **spark**, contenente i job in tale implementazione
- modulo **spark-sql**, contenente i job in tale implementazione, il job di pre processing e quelli relativi al machine learning
  - package **ml**, job relativi alla parte di machine learning