Progetto Scalable & Cloud Programming

Valentina Ferraioli, Maria Francesca Salvatore, Luca Castagnotto

Giugno 2022

1 Introduzione

Il Data mining è un processo che permette di estrarre informazioni utili a partire da una grande quantità di dati. Per fare ciò si possono utilizzare diverse tecniche che variano in base al tipo di informazione che si vuole estrarre dai dati a disposizione, come ad esempio: classificazione, clustering, association rules, outer detection, etc. In questo progetto, ci siamo soffermati a studiare come vengono implementati i sistemi di raccomandazione per i film sulla base della cronologia e delle valutazioni pregresse fornite da un utente. Una delle possibili implementazione di questi sistemi fa proprio uso delle Association Rules, per cui ci siamo concentrati a capire come funziona questa tecnica per poi analizzare nel dettaglio l'efficacia di uno degli algoritmi che utilizza per il data mining, chiamato **Apriori**.
Per procedere con l'analisi di Apriori abbiamo reimplementato l'algoritmo in linguaggio *Scala*, facendo particolare attenzione che potesse scalare in contesti paralleli e distribuiti usufruendo del framework *Spark*, che permette, appunto, di eseguire una computazione parallela e distribuita occupandosi anche della gestione delle risorse. È stata eseguita quindi un'analisi della scalabilità comparando esecuzioni eseguite su macchina locale con altre eseguite su clusters, usufruendo di *Google cloud platform*. In seguito vedremo molto brevemente come vengono utilizzate le association rules nei sistemi di raccomandazione per poi andare nel dettaglio dell'algoritmo **Apriori**. Analizzeremo poi la nostra implementazione dell'algoritmo giustificando alcune scelte implementative ed, infine, procederemo con un'analisi dei risultati ottenuti nei diversi ambienti.

2 Panoramica dell'algoritmo

2.1 Association Rules

Le association rules hanno lo scopo di catturare la relazione che intercorre tra diversi elementi in base ad eventuali pattern di co-occorrenza nella lista di transazioni.
Nel nostro caso le transazioni consistono in una tupla contentente i film visti e recensiti positivamente da un utente e l'individuazione di eventuali pattern

in queste transazioni ci permettono di fare delle affermazioni. Ad esempio, potremmo osservare che il 90% degli utenti che hanno guardato "Star Wars" e "The Empire strikes back", sono anche interessate a " A new Hope". Queste osservazioni ci permettono di consigliare " A new Hope" ad un nuovo utente, il quale ha apprezzato precedentemente i film "Star Wars" e "The Empire strikes back". Ma vediamo nel dettaglio cos è un'association rules e perchè è possibile utilizzarla per i nostri scopi.

Sia $I = \{i_1, i_2..i_n\}$ un insieme di letterali, detti *items*.
Sia D = $\{T_1, T_2, Tn\}$ un insieme di transazioni, tale che ogni transazione T è un insieme di items, detto *itemset* tale che $T \subseteq I$. Diciamo che T *contiene* $X \subseteq I$se$X \subseteq T$.
Una association rules è un'implicazione del tipo $X \Rightarrow Y$ dove $X \subseteq I, Y \subseteq I$ e $X \cap Y = 0$. La regola vale in D con *confidence* c sse $c\%$ delle transazione in D che contengono X contengono anche Y.
La regola $X \Rightarrow Y$ vale in D con *supporto* s sse $s\%$ delle transazione in D che contengono $X \cup Y$.
Dato un insieme di transazioni D, il problema di minare le association rules consiste nel generare tutte le association rules che soddifano un minimo supporto e confidenza. Possiamo decomporre il problema in due sottoproblemi:

1. Trovare tutti gli *itemsets* il quale supporto è maggiore di un *supporto minimo* specificato dall'utente. Questi itemset sono detti **frequent itemsets**.

2. Usare i **frequent itemsets** per generare le regole. L'idea generale è che se, diciamo, ABCD e AB sono frequent itemset, allora possiamo verificare che la regola $AB \Rightarrow CB$ valga calcolando

$$confidence = \frac{support(ABCD)}{support(AB)}$$

   Se $confidence \geq min\_confidence$, allora la regola è valida

Il primo sottoproblema è quello più impegnativo a livello computazionale tra i due. Per questo abbiamo deciso di focalizzarci sull'analisi di questo sottoproblema risolto molto spesso attraverso l'algoritmo **Apriori**, che si occupa del mining dei frequent itemsets. In seguito vedremo nel dettaglio una sua possibile implementazione, che sarà poi usata come base per la nostra reimplementazione dell'algoritmo.

2.2 Apriori

Notazione:

Siano:

- k-itemset → Itemset contenente k items.

- $L_k \rightarrow$ Insieme di k-itemsets frequenti (quelli con supporto minimo). Ogni elemento di L_k ha la forma (itemset, support_count)
- $C_k \rightarrow$ canditati k-itemsets (potenziali frequent itemsets). Ogni elemento di L_k ha la forma (itemset, support_count)
- $P^i \rightarrow$ Processor con Id i.
- $D^i \rightarrow$ dataset locale al processore P^i
- $DR^i \rightarrow$dataset locale al processore P^i dopo il ripartizionamento
- $C_k^i \rightarrow$ The candidate set maintained with the Processor P durinq the Mh pass (there are kitems in each candidate).

Algoritmo

Come detto precedentemente, **Apriori** si occupa di identificare tutti i frequent itemsets.
Il primo step dell'algoritmo, definito con k = 1, consiste nel contare le occorrenze di ogni elemento all'interno dell'insieme delle transazioni D, per determinare i frequent 1-itemsets. I passaggi successivi, diciamo k = k + 1 , si compongono di due fasi.

- Prima di tutto, i frequent itemsets L_{k-1}, trovati nel passo k-1, sono utilizzati per generare gli itemsets candidati C_k, utilizzando la procedura **CandidatesGeneration** descritta di seguito.
- Successivamente, il dataset viene scansionato e viene calcolato il supporto per ogni $c_i \in C_k$.

```
L_k := (frequent 1-itemsets);
k := 2;
while L_{k-1} ≠ 0 do
  begin
      # k represents the pass number
      C_k := New candidates of size k generated from L_{k-1} ;
      forall transactions t ∈ D do
        Increment the count of all candidates in C_k that are
            contained in t;
      L_k := All candidates in C, with minimum support;
      k := k + 1;
end
Answer := ∪_k L_k;
```

CandidatesGeneration. Dato L_{k-1}, ovvero l'insieme di tutti i frequent (k - 1)-itemsets, vogliamo generare dell'insieme di tutti i frequent k-itemsets.
L'intuizione dietro la procedura di *CandidatesGeneration* di Apriori è che se un itemset X soddisfa un supporto minimo, allora lo soddisferanno tutti i sottoinsiemi di X. La generazione di candidati è divisa in due fasi:

- **fase di joining**, unisci $L_{k-1-}eL_{k-1}$

```
insert into C_k,
select p.item_1, qitem_2, ..., p.item_{k-l}, q.item_{k-1},
from L_{k-1}p, L_{k-1}q
where pitem_1 = qitem_1, ..., pitem_{k-2} = qitem_{k-2}, pitem_{k-1} < qitem_{k-1};
```

- **fase di pruning**, elimina tutti gli itemsets $c \in C$,tali che un certo (k - 1)-sottoinsieme di c non è in L_{k-1}.

Per esempio, sia $L_3 = \{\{123\}, \{124\}, \{134\}, \{135\}, \{2341\}\}$. Dopo il passo join, $C_4 = \{\{1234\}, \{1345\}\}$. Il passo di pruning consisterà nel cancellare l'itemset $\{1345\}$ perché l'itemset $\{145\}$ non è in L_3. Per cui soltanto $\{1234\} \in C_4$.

2.3 Versione parallela: Count Distribution

Questa versione dell'algoritmo è stata presa da [AS96] ed è pensata per un'architettura *shared-nothing*, dove ciascuno dei processori N ha memoria privata e disco privato. I processori sono collegati da una rete di comunicazione nel quale è possibile utilizzare *message passing* e i dati sono distribuiti uniformemente sui dischi di ogni processore, garantendo che ogni processore abbia un numero uguale di transazioni.
Nel primo passaggio, ogni processore P^i genera dinamicamente il suo local candidate itemset C_1^i a seconda delle transazioni presenti nella sua partizione di dati locale D^i. Quindi, i candidati ottenuti dai diversi processori possono non coincidere, per questo è necessario che vengano scambiati per determinare C_k globale.
Per tutti gli altri passaggi $k \geq 1$, l'algoritmo invece funziona come segue:

1. Ogni processoreP^i genera il C_k completo, a partire da L_{k-1} ottenuto dallo step precedente. Osserviamo che poiché ogni processore ha lo stesso L_{k-1}, genererà lo stesso identico C_k.

2. Processore P^i scansiona la sua partizione di dati D^i e calcola il support count locale per i candidati in C_k

3. Processore P^i scambia il C_k ottenuto localmente con tutti gli altri processori per ottenere C_k globale. I processori sono costretti a sincronizzarsi in questa fase.

4. Ogni processore P^i ora calcola L_k partendo da C_k

5. Ogni P^i decide autonomamente se fermarsi o continuare. La decisione sarà identica perchè tutti i processori dispongono dello stesso L_k

In questo modo ad ogni passaggio i processori possono eseguire la scansione dei dati locali in maniera asincrona ed in parallelo, per poi sincronizzarsi alla fine di ogni passaggio per ottenere i conteggi globali.

3 Implementazione

L'algoritmo Apriori è stato implementato nella versione *Count Distribution*, utilizzando il linguaggio **Scala** (versione 2.12.14) e la piattaforma **Spark** (versione 3.1.2).

3.1 Gestione degli input

I test sono stati eseguiti su due dataset forniti da GroupLens:

- MovieLens 100K Dataset: composto da 100 000 valutazioni su 9 000 film da parte di 600 utenti,
- MovieLens 20M Dataset: composto da 20 000 000 valutazioni su 27 000 film da parte di 138 000 utenti.

Per prima cosa abbiamo scartato tutti i film aventi una valutazione inferiore a 3 stelle, così che i frequent itemsets fossero composti esclusivamente dai film apprezzati, e che possano ottenere buoni feedback se consigliati. Poi abbiamo modellato le transazioni come key-value RDD della forma (*user, moviesList*), il che ci permette di sfruttare al massimo la parallelizzazione nelle fasi successive dell'algoritmo.
Lavorare su un key-value RDD infatti ci permette di fare una ripartizione customizzata, che permette di minimizzare lo shuffling durante l'esecuzione di operazioni di reduce in ambiente distribuito, aumentandone così le performance.
Nella nostra implementazione abbiamo partizionato l'RDD in n partizioni attraverso la funzione *HashPartitioner*, dove n corrisponde al numero di core disponibili nell'ambiente di test.

3.2 Count Distribution

L'algoritmo è stato implementato usando come base la definizione vista in 2.3, con delle piccole variazioni:

- Master genera C_k, ottenuto da L_{k-1}.
- Processore P^i scansiona la sua partizione di dati D^i e calcola il support count locale per i candidati in C_k
- Master raccoglie i local_support_count di ogni P^i, attraverso una *ReduceByKey*
- Master calcola L_k sulla base dei risultati raccolti
- Master decide se continuare o fermarsi.

Le variazioni rimuovono alcune operazioni che nella versione parallela venivano eseguite su ogni singola partizione, portando agli stessi risultati, quali:

- calcolo di C_k a partire dallo stesso L_{k-1}.
- calcolo di L_k a partire da C_k .

In altre parole, questo tipo di operazioni abbiamo deciso di farle eseguire direttamente al nodo master una sola volta e di procedere con la comunicazione di C_k ed L_k al momento opportuno.

```
# K = 1
#ds_full =  (userId, movieId)
val transactions = ds_full.aggregateByKey(List[String]())((acc, x)
    => x :: acc, (acc1, acc2) => acc1 ::: acc2).map(t =>
    t._2.map(_.toInt))
# (movieId,1) -> (movieId, n_of_occurrences) -> keep just the
    movieIds that satifsy min support
val frequent_singleton = transactions.flatMap(
    transaction => transaction.
    map(movieId => (movieId, 1))).
    reduceByKey((x, y) => x + y).
    filter(x => x._2 >= min_support).
    map(z => List[Int](z._1)).
    collect().
    toList
var last_frequent_itemsets =
    sort(frequent_singleton).toList.map(_.toList)
var frequent_itemsets = ListBuffer[List[Int]]()
frequent_itemsets ++= last_frequent_itemsets
```

Calcolo dei frequent Itemset per k = 1, che come abbiamo visto nella panoramica dell'algoritmo viene trattato in modo particolare.

```
#k > 2
    var k = 2
    while(last_frequent_itemsets.nonEmpty) {
      val candidates =
          candidates_generation(last_frequent_itemsets, k)
          #calcolo dei candidati

      val candidatesPartitions = transactions
                                .mapPartitions(x =>
                                    local_support_count(x,
                                    candidates, k))

      val new_frequent_itemsets = candidatesPartitions
                                .reduceByKey((x,y) => x + y)
                                .filter(z => z._2 >= min_support)
       #collect itemset count of each partitions, and get the
           global count, then delete all of them that do not
           satisfy the min_support
      last_frequent_itemsets = sort(new_frequent_itemsets
                                .map(x => x._1.sorted)
                                .collect().toList).map(_.toList)
      frequent_itemsets ++= last_frequent_itemsets
      k = k + 1
    }
```

È in questo punto che abbiamo una reale esecuzione distribuita; infatti dopo il calcolo dei candidati l'RDD di transazioni viene partizionato tra i nodi attraverso la funzione *mapPartitions*. Ogni nodo si occupa di calcolare quante volte ciascun candidato compare nella porzione di dati a sua disposizione. Poi con una *reduceByKey* sommiamo tali numeri e filtriamo i candidati che raggiungono il supporto minimo.

```
def local_support_count(transactions: Iterator[Iterable[Int]],
    candidates: Set[Set[Int]], k: Int) : Iterator[(List[Int],
    Int)] = {
      var transactionSet = new ListBuffer[Set[Int]]()
      for(transaction <- transactions){
        transactionSet += transaction.toSet
      }
      #keep all elements in candidates that exist as a subset of
          transaction
      val filteredItemsets = candidates.map{
        itemset => (itemset, transactionSet.count(transaction =>
            itemset.subsetOf(transaction)))
      }.filter(x => x._2 > 0).map(x => (x._1.toList, x._2))
      filteredItemsets.iterator
    }

def candidates_generation(last_frequent_itemsets: List[List[Int]],
    k : Int) : Set[Set[Int]] = {
    # joining
    val it1 = last_frequent_itemsets.iterator
    var candidates = Set[Set[Int]]()
    while (it1.hasNext) {
      val item1 = it1.next()
      val it2 = last_frequent_itemsets.iterator
      while(it2.hasNext) {
        val lNext = it2.next()
        if(item1.take(k - 2) == lNext.take(k - 2) && item1 !=
            lNext && item1.last < lNext.last) {
          val l = List(item1 :+ lNext.last)
          val lF = l.flatten
          candidates += lF.toSet
        }
      }
    }
    # pruning
    val last_freq_itemset_Set =
        last_frequent_itemsets.map(_.toSet).toSet
    if(k > 2) {
      for(candidate <- candidates) {
        for(subset <- candidate.subsets(k - 1)) {
          if(!last_freq_itemset_Set.contains(subset)) {
            candidates -= candidate
          }
        }
      }
    }
    candidates
  }
```

La *candidates_generation* è costruita come descritto al punto 2.2 e viene eseguita ad ogni iterazione dell'algoritmo dal Master. La funzione *local_support_count* viene invece eseguita in maniera separata sui singoli nodi, in modo tale che ognuno calcoli il supporto di tutti i candidati rispetto alla propria porzione di transazioni.

3.3 Ambienti di test

L'algoritmo è stato testato analizzando le performance al variare di tre variabili principali:

- *min_support* = $\{7, 5\%, 10\%, 12\%, 14\%, 16\%, 18\%, 20\%\}$ che influisce sulla complessità della computazione
- tipo di ambiente in cui eseguire l'algoritmo: *locale* o *cloud*, che definisce il tipo di architetture al quale si ha accesso. Nel caso in cui si voglia testare in locale infatti abbiamo a disposizione N_Nodi_Worker = 1, e N_vCPUs = N_ofCores, mentre nel caso in cui si stia lavorando su cloud, le due variabili devono essere inizializzate tenendo in considerazione il cluster sul quale si sta lavorando. In base all'architettura sottostante infatti, definiamo il numero di partizioni, N_partizioni, in cui è conveniente partizionare l'RDD al fine di ottenere le performance migliori.
- N_partizioni: = N_Nodi_Worker * N_vCPUs_Per_Worker

I nostri ambienti di test sono stati i seguenti

- locale su una macchina con processore octa-core, da cui abbiamo definito N_Nodi_Worker = 1, e N_vCPUs = 8, N_partizioni = 8.
- su cloud abbiamo provato creando cluster di diverso tipo, ovvero:
  - cluster con 1 nodo master e 2 nodi worker, ognuno avente 4 vCPUs, da cui N_partizioni = in 8
  - cluster con 1 nodo master e 4 nodi worker, ognuno avente 4 vCPUs, da cui N_partizioni = 16

4 Risultati

L'analisi dei risultati si basa sui tempi di esecuzione dell'algoritmo Apriori al variare della dimensione del dataset, del valore del minimo supporto, della macchina (locale o cloud) e delle sue risorse.

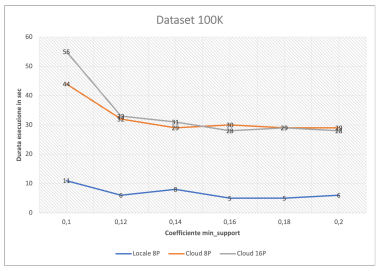

Figura 1

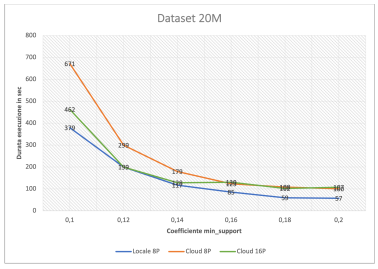

Figura 2

Figura 3

La prima cosa evidente è come i tempi di esecuzione per l'analisi dei due dataset abbiano ordini di grandezza diversi. Infatti, mentre per l'analisi del *Dataset 100k* necessitiamo al massimo di 56 sec per eseguire apriori con *min_support* = 0.1, per l'analisi del *Dataset 20M* registriamo un valore massimo di 671 sec. Questa differenza è dovuta al tempo necessario per analizzare i dataset.
Notiamo poi un trend comune ai due grafici che consiste nel decrescere dei tempi di esecuzione all'aumentare del coefficiente di *min_support*, questo perchè all'aumentare del min_support alziamo la threshold sopra la quale consideriamo i film "frequenti", andando di conseguenza ad eliminare da subito gran parte degli itemset e quindi diminuendo il numero di iterazioni dell'algoritmo necessarie.
Infine, osservando i tempi di esecuzione ottenuti dall'analisi dei due dataset sui diversi ambienti e sui vari coefficienti possiamo considerare due casi interessanti:

- ambiente **cloud16P**. In figura 1 (100k) vediamo che ottiene le performance peggiori rispetto agli altri ambienti. Questo risultato ci sembra ragionevole in quanto stiamo lavorando su un dataset relativamente piccolo, per cui procedere con la distribuzione dei dati in 16 partizione risulta inutile. Al contrario in figura 2 (20M), l'ambiente **cloud16P** ottiene performance migliori rispetto a **cloud8P** in quanto il tempo consumato per la distribuzione dei dati viene ripagato dalla distribuzione del workload.

- ambiente **local8P**, in entrambe le figure vediamo che le performance in locale risultano migliori per l'analisi di entrambi i dataset

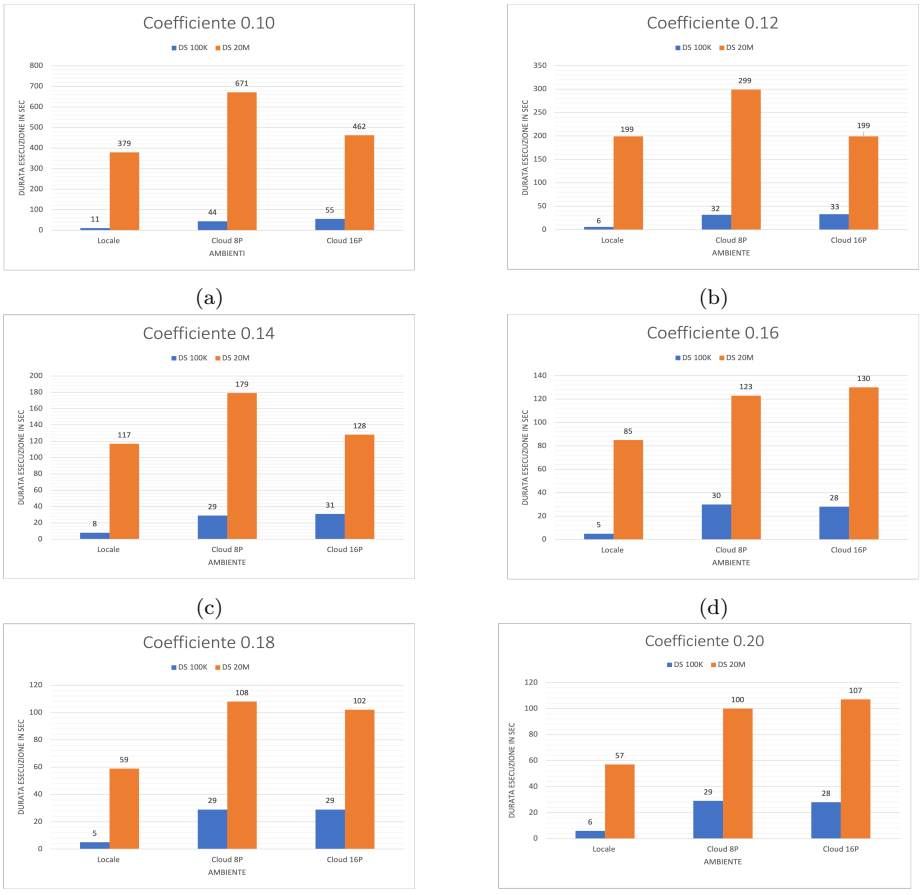

Figura 4: Tempi di esecuzione per ciascun coefficiente

Gli stessi risultati discussi precedentemente si possono osservare anche nei grafici in figura 4, dove si vuole fare un confronto delle performance ottenute per ogni *min_support* in ciascun ambiente. Per prima cosa notiamo nuovamente che le performance ottenute in locale sono le migliori, poi rileviamo due altri fatti:

- per *Dataset 100k* osserviamo un lieve miglioramento con coefficiente di *min_support*= 0.1 su ambiente cloud 8P rispetto a cloud16P, tuttavia successivamente non si notano differenze sostanziali nelle performance ottenute su cloud, in quanto registriamo un distacco di pochi secondi che potrebbe essere dovuto alla carico di rete
- per *Dataset 20M* invece, notiamo che si hanno risultati migliori con cloud16P rispetto a cloud8P per coefficiente pari a {0,1, 0.12, 0.14}, mentre per i coefficienti successivi abbiamo risultati inversi. Inoltre, per *Dataset 20M*,

comparando le performance su cloud e locale per i coefficieni di min support = {0,1, 0.12, 0.14}, non abbiamo un distacco netto come quello registrato su *Dataset 100k*; in particolare, per *min_support* = 0.12 abbiamo un lo stesso tempo di esecuzione in locale e cloud pari a 199sec, mentre per *min_support* = 0.14 abbiamo 117 sec in locale e 128 in cloud16P, con uno scarto di soli 10sec.
Questo è un risultato interessante. Infatti, lavorando su un dataset di dimensione medio piccole non riusciamo a sfruttare appieno i benefici della distribuzione, tuttavia , qui possiamo vedere come situazioni in cui sia necessaria una computazione più pesante, riusciamo ad ottenere in cloud performance equivalenti a quelle ottenute in locale, probabilmente migliori se si lavorasse su un dataset di dimensione maggiore.

5 Conclusione

Le multiple esecuzioni della nostra versione dell'algoritmo Apriori Count Distribution rivelano che i dataset a nostra disposizione sono troppo piccoli affinché si possa ottenere un vantaggio dal lavorare su cloud. La maggior distribuzione del lavoro non è tale da sopperire il costo di comunicazione tra i nodi della rete del cluster Dataproc, per cui le prestazioni in locale rimangono le migliori, solo in un paio di casi si è riusciti ad ottenere in cloud le stesse performance ottenute in locale. Il limite più considerevole è rappresentato dall'operazione di *collect* utile a creare la lista degli ultimi frequent itemsets calcolati, che servono per la generazione dei candidati dell'iterazione successiva.
Tuttavia, il nostro progetto è da considerarsi solo una possibile implementazione in versione distribuita dell'algoritmo **Apriori** nella sua versione parallela **Count distribution**, ma ci sono diverse ottimizzazioni che potrebbero essere apportate a questa versione per ottenere performance migliori. Inoltre, sottolineiamo che ci siamo soffermati sull'implementazione di Apriori, in quanto viene considerato il sottoproblema più costoso nell'utilizzo di association rules, tuttavia un' aggiunta futura e necessaria a questo progetto consiste nell'applicazione delle association rules una volta ottenuti i frequent_itemset con l'utilizzo di Apriori, in modo tale da offrire all'utente il film più adatto ai suoi gusti sulla base della cronologia dei film che gli sono piaciuti.

Bibliografia

[AS96] Rakesh Agrawal e John C Shafer. ≪Parallel mining of association rules≫. In: *IEEE Transactions on knowledge and Data Engineering* 8.6 (1996), pp. 962–969.