UNIVERSITÀ
degli STUDI
di CATANIA

Relazione Analisi e Gestione dei Dati

Algoritmi per la ricerca di Insiemi Frequenti su Dataset

Mirko R. Aiello W82/000045
Università degli Studi di Catania - A.A. 2015-2016

Sommario

1 - Obiettivi

Il progetto in analisi, inquadrato all'interno del Corso di Analisi e Gestione dei Dati, presenta i seguenti obiettivi:

- Studio preliminare del design delle classi, atto alla realizzazione di una libreria.
- Implementazione degli algoritmi per la ricerca di insiemi frequenti mediante il linguaggio di programmazione C#.
- Analisi dei risultati e dei tempi di esecuzione.

Si è utilizzato L'IDE **Visual Studio 2015 Community Edition** per l'implementazione, e il prompt dei comandi (**CMD**) per l'esecuzione.
Sono stati inoltre utilizzati alcuni dataset reperibili al seguente link : http://fimi.ua.ac.be/data/.

La strumentazione utilizzata, infine, è la seguente :

PC FISSO

Intel(R) Core(TM) i5-6600K CPU @ 4.50 GHz
RAM 8,00 GB DDR4 @ 2666
Tipo di SO: 64 bit
SO: Windows 10 Pro

PC PORTATILE

Dell Inspiron N5110

1.1 - Concetti Base

TID	Items
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer
5	Bread, Milk, Diaper, Coke

	Beer	Bread	Milk	Diaper	Eggs	Coke
T_1	0	1	1	0	0	0
T_2	1	1	0	1	1	0
T_3	1	0	1	1	0	1
T_4	1	1	1	1	0	0
T_5	0	1	1	1	0	1

Itemset: insieme di uno o più prodotti.
Supporto (assoluto) di *X*: frequenza di occorrenza dell'itemset ***X***.
Supporto (relativo), *s*, frazione di transazioni che contengono X (es., la probabilità che una transazione contiene X).

Un itemset X è **frequente** se il supporto di X è maggiore o uguale ad un valore *threshold* chiamato minsup.

1.2 - Linguaggio di programmazione C#

Il **C#** è un linguaggio di programmazione orientato agli oggetti sviluppato da Microsoft all'interno dell'iniziativa .NET, e successivamente approvato come standard ECMA.
La sintassi del C# prende spunto sia da quella di Delphi (hanno il medesimo autore, ovvero Anders Hejlsberg), di **C++**, di **Java** e di **Visual Basic** per gli strumenti di programmazione visuale e per la sua semplicità.
Il risultato è un linguaggio con meno simbolismo rispetto a C++, meno elementi decorativi rispetto a Java, ma comunque orientato agli oggetti in modo nativo.

2.1 - Apriori

L'algoritmo procede per passi, a livelli :

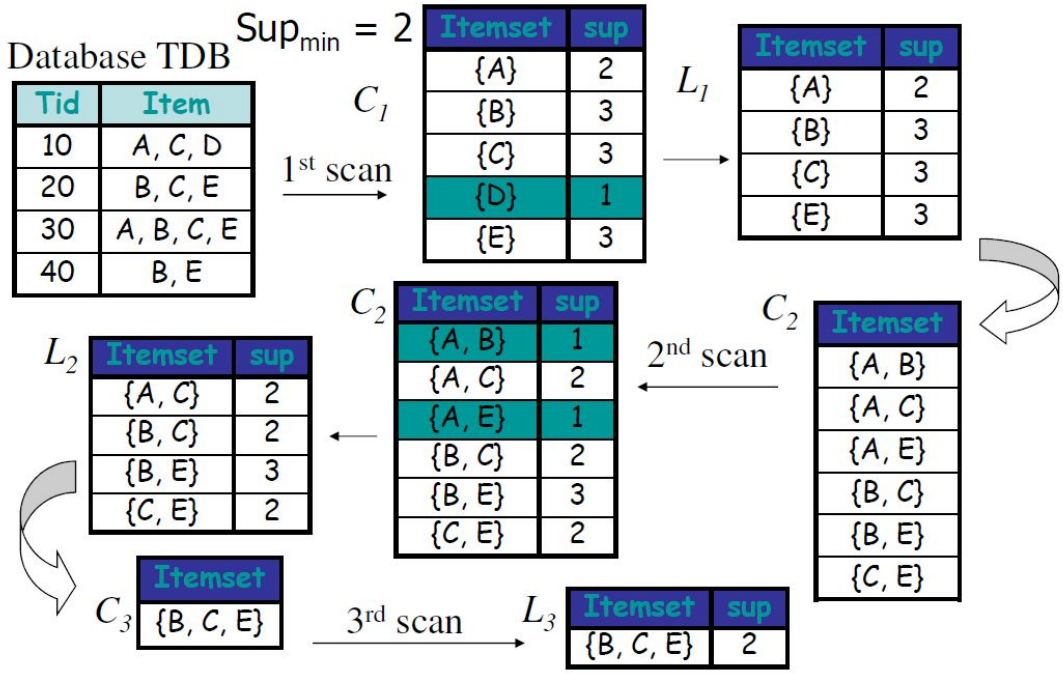
> $Read(s)$;
> $L_1 = \{ a \mid a\ appare\ con\ frequenza >= s \}$
> $L_2 = \{ \{a,b\} \mid a,b\ \varepsilon\ L1\ appare\ con\ frequenza >= s \}$
> $L_3 = \{ \{a,b,c\} \mid \{a,b\},\ \{a,c\},\ \{b,c\}\ \varepsilon\ L_2\ \{a,b,c\}\ appare\ con\ frequenza >= s\}$

e così via. Questa è invece la sua versione in pseudocodice :

C_k: Itemset candidati di lunghezza k
L_k: Itemset frequenti di lunghezza k
s: supporto minimo

L_1 = {item frequenti};
for (k = 1; L_k !=∅; k++) do begin
  C_{k+1} = Candidati generati da L_k;
  for each transazione t nel database do
    incrementa il conteggio per tutti i candidati in in C_{k+1} che sono contenuti in t
  L_{k+1} = candidati in C_{k+1} con supporto minimo s
end
return $\cup_k L_k$;

E successivamente un esempio dell'algoritmo in funzione :



Si procede per un numero fissato di iterazioni oppure fino a quando Li diventa vuoto (se vogliamo trovare l'insieme massimale).

2.2 - PCY (Park, Chen, Yu)

L'idea base del PCY è quella di usare una tabella *hash* per eliminare alcune coppie che non possono essere frequenti durante il calcolo di L_1. Di fatto, è un'ottimizzazione dell'algoritmo Apriori.
Il passo 1 è il seguente :

```
FOR each transaction t in T DO
   FOR each item in t DO
       add 1 to item's count
   END FOR
   FOR earch pair of items in t DO
       Hash the pair to a bucket and add 1 to the count for that bucket
   END FOR
END FOR
```

Mentre il passo 2 considera tutte le potenziale coppie (i, j) e verifica che i e j, presi individualmente, siano item frequenti, e che il bucket nella tabella hash contenente (i, j) abbia un valore maggiore al supporto *s*. Queste due condizioni sono **necessarie** ma non sufficienti per considerare una coppia un insieme frequente.
Infine, le coppie sopravvissute dovranno avere un supporto maggiore ad s relativamente all'intero insieme.

3 - Approccio mediante algoritmi randomizzati

Gli algoritmi discussi precedentemente utilizzano un passo, qualunque sia la grandezza del dataset.
Se la memoria principale è troppo piccola non possiamo fare nulla per evitare i k passi per ottenere l'insieme degli insiemi frequenti.
Qui di seguiti si tratteranno algoritmi che riescono a risolvere il problema in almeno 2 passi.

3.1 - Approccio Semplice (Sampling)

L'algoritmo Sampling è diviso in 2 passi : nel passo 1 viene selezionato un campione random di carrelli, sulla base di una distribuzione **fissata** di probabilità. Nel passo 2, viene utilizzato uno degli algoritmi descritti precedentemente, sul campione ottenuto, per trovare gli insiemi frequenti. Il supporto *s* viene corretto di conseguenza per adattarsi alla nuova dimensione del campione.
Per esempio, se scegliamo un campione pari all'1% del carrello, il supporto *s* dovrà essere pari a *s/100*.
Questo algoritmo è molto veloce, tuttavia, dato che il campione considerato è un sottoinsieme del dataset originale, è possibile che produca *falsi negativi* (itemset frequenti nel complesso ma non nel sottoinsieme) e *falsi positivi* (itemset frequenti nel sottoinsieme ma non nel complesso).

3.2 - Savasere-Omiecinski-Navathe (SON)

L'algoritmo SON cerca di migliorare il Sampling, eliminando i falsi positivi e negativi che si vengono a creare. L'idea è di dividere il dataset di input in ***chunks*** (pezzi) più piccoli di uguale grandezza. Quindi, si esegue uno degli algoritmi precedentemente trattati (Sampling, Apriori, PCY) per ognuno dei chunks così ottenuti.
Useremo *ps* come soglia, dove *p* è frazione della dimensione del chunks rispetto all'intero file, mentre *s* è il supporto.
Una volta che tutti i chunks sono stati processati, considereremo l'unione di tutti gli itemset trovati come **candidati**.
Tenendo presente che il numero di chunks è *1/p*, se un itemset non è frequente in almeno 1 chunk, allora il suo supporto sarà inferiore a *ps* in ogni chunk. Possiamo quindi osservare che *(1/p) ps* < *s*, quindi l'insieme non è frequente.
Inoltre, ogni oggetto che è frequente nell'intero insieme lo sarà in almeno 1 chunk, quindi non ci saranno *falsi negativi*.
Riassumendo, il passo 1 dell'algoritmo processa ogni singolo chunk leggendo tutto il dataset. Nel secondo passo, l'insieme di tutti gli itemset dovrà essere filtrato scegliendo come candidati tutti gli elementi che avranno un supporto di almeno *s*.

3.3 Toivonen

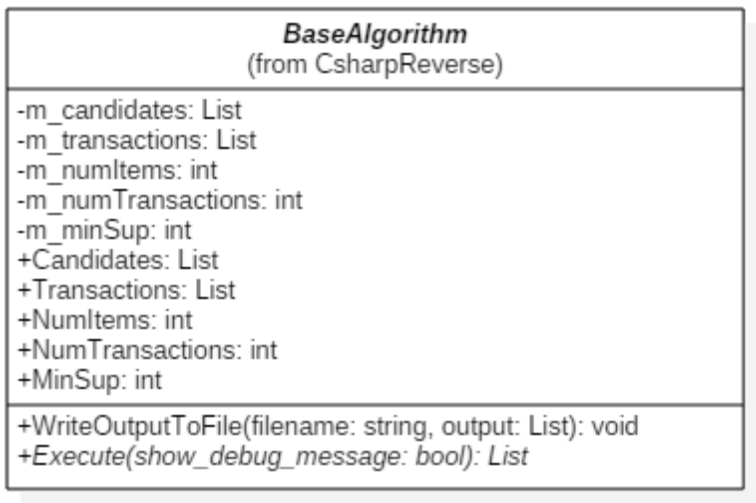
L'algoritmo Toivonen seleziona un campione random di carrelli che entri in main memory, applicando il metodo A-Priori ma con minimo supporto diminuito in modo che non sfuggano insiemi frequenti (ad es. se il campione è 1% usa s/125). Si aggiunge quindi ai candidati la *frontiera negativa*, quegli insiemi di articoli S che non sono risultati frequenti,nel campione, ma tali che ogni

loro sottoinsieme massimale lo è. Ad esempio {A,B,C} è risultato non frequente ma {A,B} , {A,C} e {B,C} sono risultati frequenti nel campione.
Viene effettuato quindi un controllo sui candidati. Se effettivamente nessun elemento della frontiera negativa è frequente nella totalità dei dati allora gli insiemi frequenti sono esattamente i candidati che superano la soglia. Se invece sfortunamente qualche elemento della frontiera negativa supera la soglia allora non sappiamo se un qualche suo soprainsieme possa essere frequente e quindi dobbiamo ripetere il processo (oppure accettiamo l'ipotesi di avere alcuni falsi negativi).

4 - Diagramma delle classi

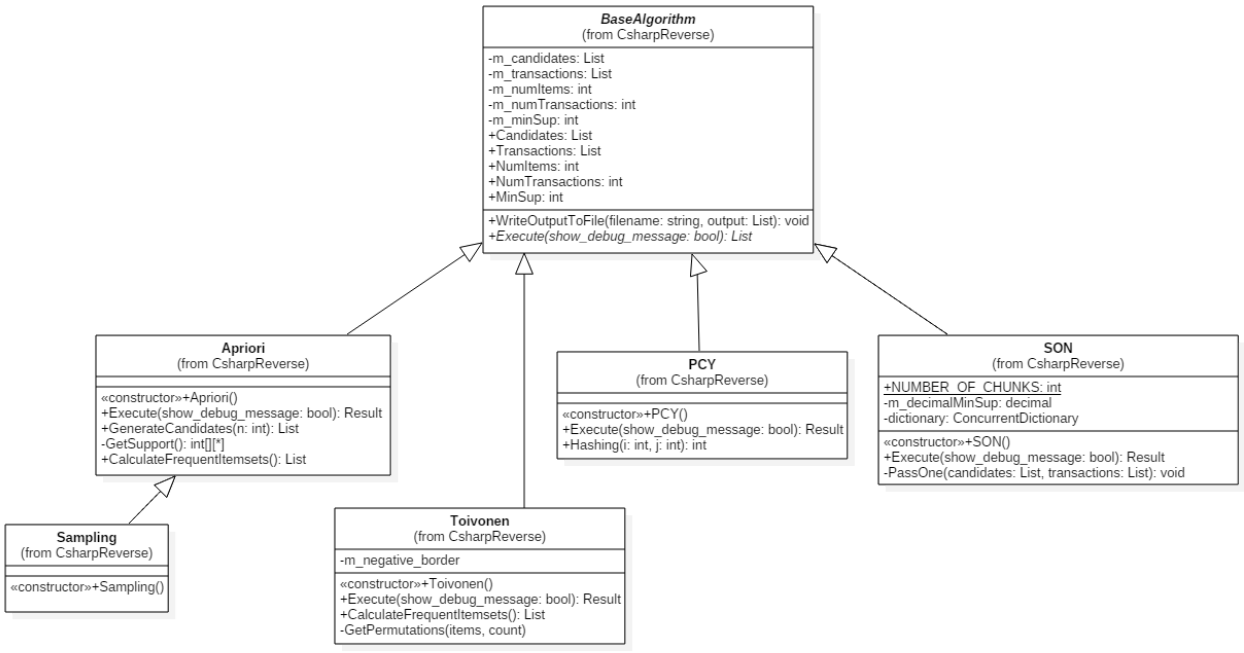
In questo paragrafo verrà discussa la struttura delle classi e il loro funzionamento con relativa implementazione.

4.1 - BaseAlgorithm



L'idea base dietro queste classi è quella di fornire all'utente finale una struttura che permetta di implementare nuovi algoritmi per la ricerca di insiemi frequenti in maniera facile e veloce.

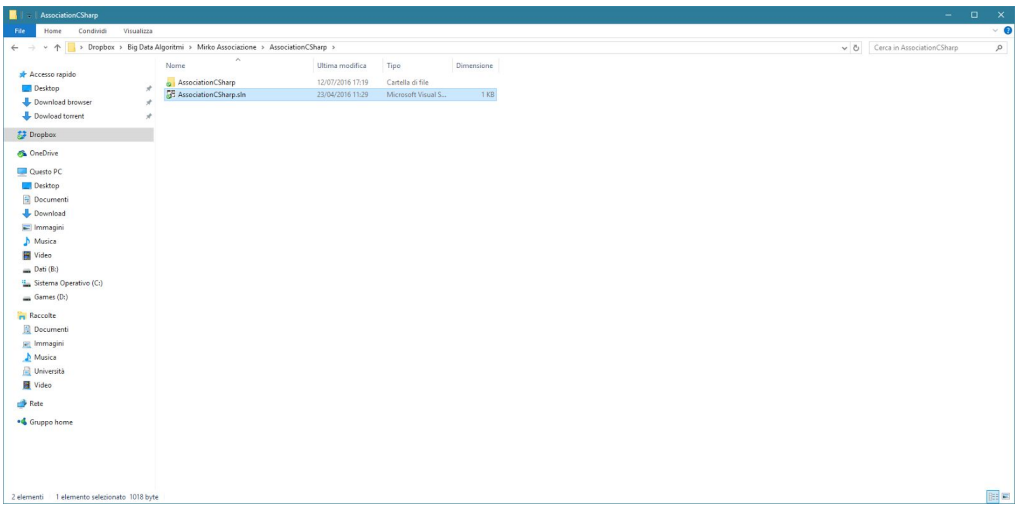
La classe astratta *BaseAlgorithm* contiene 5 proprietà : **Candidates** (la lista dei candidati), **Transactions** (la lista delle transazioni), NumItems (il numero univoco di oggetti) **NumTransactions** (il numero delle transazioni) e **MinSup** (il minimo supporto, espresso in funzione del numero delle transazioni),
Ogni sottoclasse inoltre, dovrà implementare il metodo ***Execute***, che rappresenta l'esecuzione dell'algoritmo con relativo output (una List di **SupportElement**, classe che contiene due prorietà: **Label**, una stringa che rappresenta l'elemento, e **Count**, un intero che rappresenta il suo supporto).
Tale classe, infine, si occuperà anche dell'eventuale output mediante il metodo **WriteOutputToFile**, a cui verrà passata una stringa che contiene il nome del file che si vuole scrivere, e la lista di output.



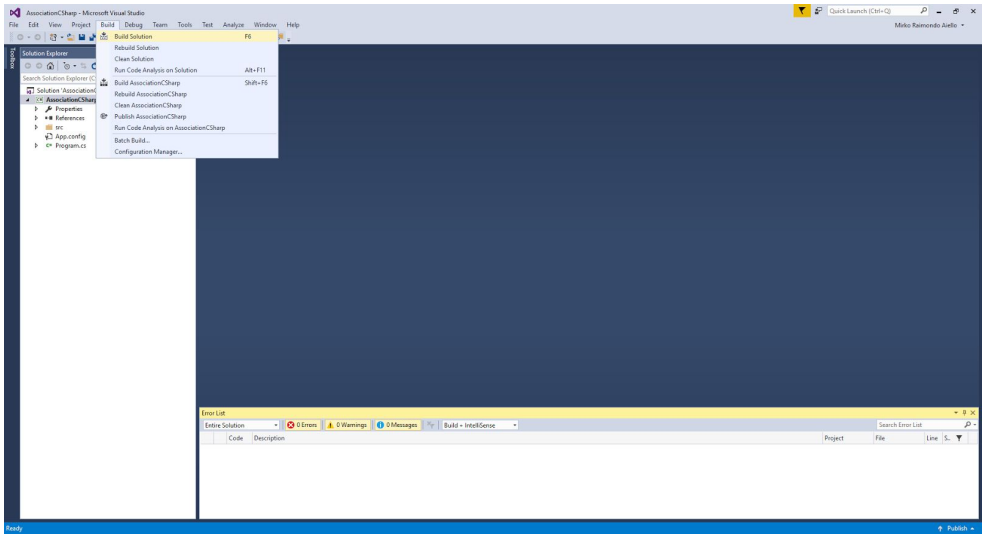
Passiamo infine all'implementazione vera e propria, ogni sottoclasse di **BaseAlgorithm** dovrà effettuare l'overriding del metodo ***Execute***.
I metodi rimanenti sono di supporto allo specifico algoritmo.
La classe Apriori conterrà i metodi per la generazione dei candidati (**GenerateCandidates**, dove n rappresenta gli elementi contenuti in ogni candidato, coppie, triple...ecc), il calcolo degli insiemi frequenti a partire dai candidati (**CalculateFrequentItemset**, che restituisce la lista dei candidati) e dei valori di supporto (**GetSupport**, che restituisce un array che contiene i valori interi dei supporti).
La classe PCY contiene il metodo **Hashing**, che restituisce il valore di hash a partire da 2 indici. Il SON contiene i metodi **PassOne**, che rappresenta il passo 1 dell'algoritmo (descritto in precedenza). Infine, la classe Toivonen conterrà il metodo che permetterà la creazione del sottoinsieme massimale.

5 - Compilazione e utilizzo del programma

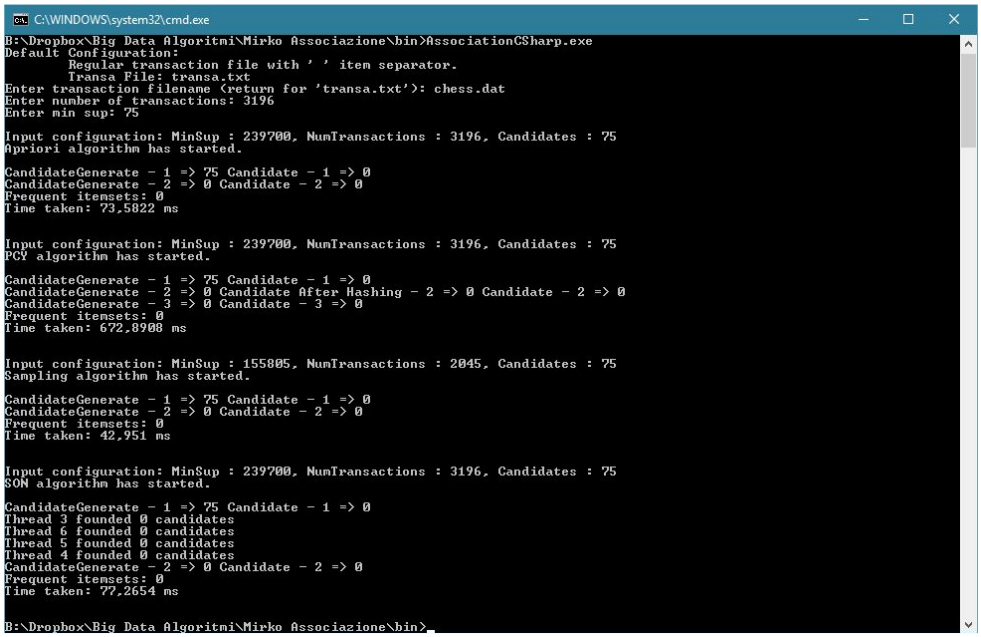
Per la compilazione del progetto, che è possibile trovare all'interno della cartella **source** è necessaria la versione 2015 o superiore di Visual Studio. Il file di progetto è denominato **FrequentItemset.sln**. Facendo doppio click su tale file si avvierà Visual Studio permettendone la compilazione.



Per compilare è sufficiente cliccare su Build->Build Solution. I file compilati si troveranno nella cartella **FrequentItemset/bin/Release** se è stata selezionata l'opzione Release, altrimenti saranno nella cartella **FrequentItemset/bin/Debug** se è stata selezionato l'opzione Debug.



Per avviare i file precompilati è sufficiente avviare il programma **FrequentItemset.exe** utilizzando la riga di comando. Dopodichè si dovranno specificare i parametri del programma che sono rispettivamente il nome del file che contiene le transazioni, il numero di transazioni e il minimo supporto (espresso in un valore decimale compreso tra 0 e 1).


C:\WINDOWS\system32\cmd.exe
B:\Dropbox\Big Data Algoritmi\Mirko Associazione\bin>AssociationCSharp.exe
Default Configuration:
Regular transaction file with ' ' item separator.
Transa File: transa.txt
Enter transaction filename (return for 'transa.txt'): chess.dat
Enter number of transactions: 3196
Enter min sup: 75
Input configuration: MinSup : 239700, NumTransactions : 3196, Candidates : 75
Apriori algorithm has started.
CandidateGenerate - 1 => 75 Candidate - 1 => 0
CandidateGenerate - 2 => 0 Candidate - 2 => 0
Frequent itemsets: 0
Time taken: 73,5822 ms
Input configuration: MinSup : 239700, NumTransactions : 3196, Candidates : 75
PCY algorithm has started.
CandidateGenerate - 1 => 75 Candidate - 1 => 0
CandidateGenerate - 2 => 0 Candidate After Hashing - 2 => 0 Candidate - 2 => 0
CandidateGenerate - 3 => 0 Candidate - 3 => 0
Frequent itemsets: 0
Time taken: 672,8908 ms
Input configuration: MinSup : 155805, NumTransactions : 2045, Candidates : 75
Sampling algorithm has started.
CandidateGenerate - 1 => 75 Candidate - 1 => 0
CandidateGenerate - 2 => 0 Candidate - 2 => 0
Frequent itemsets: 0
Time taken: 42,951 ms
Input configuration: MinSup : 239700, NumTransactions : 3196, Candidates : 75
SON algorithm has started.
CandidateGenerate - 1 => 75 Candidate - 1 => 0
Thread 3 founded 0 candidates
Thread 6 founded 0 candidates
Thread 5 founded 0 candidates
Thread 4 founded 0 candidates
CandidateGenerate - 2 => 0 Candidate - 2 => 0
Frequent itemsets: 0
Time taken: 77,2654 ms
B:\Dropbox\Big Data Algoritmi\Mirko Associazione\bin>

6. Risultati

Ogni algoritmo verrà valutato in base ai tempi di esecuzione (espressi in ms). I dataset utilizzati saranno : **chess.dat** e **mushroom.dat**, reperibili al seguente link : http://fimi.ua.ac.be/data/.
Sono stati utilizzati rispettivamente 0.7, 0.8 e 0.9 come minimo supporto per il dataset *chess.dat*, mentre per il dataset *mushroom.dat* i valori erano 0.5, 0.4 e 0.3.

5.1 - Dataset *Chess.dat*

Qui di seguito sono riportati i risultati utilizzando il dataset *chess.dat* con minimo supporto pari a **0.9**.

Algoritmo	Minimo Supporto	Itemset Frequenti	Execution Time (ms)
Apriori	2876	628	1959
PCY	2876	628	2276
Sampling	287	593	1088
SON	2876	628	***820***
Toivonen	287	628	1110

Qui di seguito sono riportati i risultati utilizzando il dataset *chess.dat* con minimo suporto pari a **0.8**.

Algoritmo	Minimo Supporto	Itemset Frequenti	Execution Time (ms)
Apriori	2556	8282	30352
PCY	2556	8282	28383
Sampling	255	8282	24500
SON	2556	8282	***17994***
Toivonen	255	8264	24506

Qui di seguito sono riportati i risultati utilizzando il dataset *chess.dat* con minimo supporto pari a **0.7**.

Algoritmo	Minimo Supporto	Itemset Frequenti	Execution Time (ms)
Apriori	2237	48969	360140
PCY	2237	48969	359956

Sampling	223	37617	***267500***
SON	2237	48969	363640
Toivonen	223	48478	328017

5.2 - Dataset *Mushroom.dat*

Qui di seguito sono riportati i risultati utilizzando il dataset *mushroom.dat* con minimo supporto pari a **0.5**.

Algoritmo	Minimo Supporto	Itemset Frequenti	Execution Time (ms)
Apriori	4062	153	1157
PCY	4062	153	1405
Sampling	406	147	***65***
SON	4062	153	460
Toivonen	324	509	68

Qui di seguito sono riportati i risultati utilizzando il dataset *mushroom.dat* con minimo supporto pari a **0.4**.

Algoritmo	Minimo Supporto	Itemset Frequenti	Execution Time (ms)
Apriori	3249	565	5412
PCY	3249	565	3447
Sampling	324	557	3005
SON	3249	565	***2003***
Toivonen	324	509	3019

Qui di seguito sono riportati i risultati utilizzando il dataset *mushroom.dat* con minimo supporto pari a **0.3**.

Algoritmo	Minimo Supporto	Itemset Frequenti	Execution Time (ms)
Apriori	2437	2735	22301
PCY	2437	2735	14247
Sampling	243	2735	16711

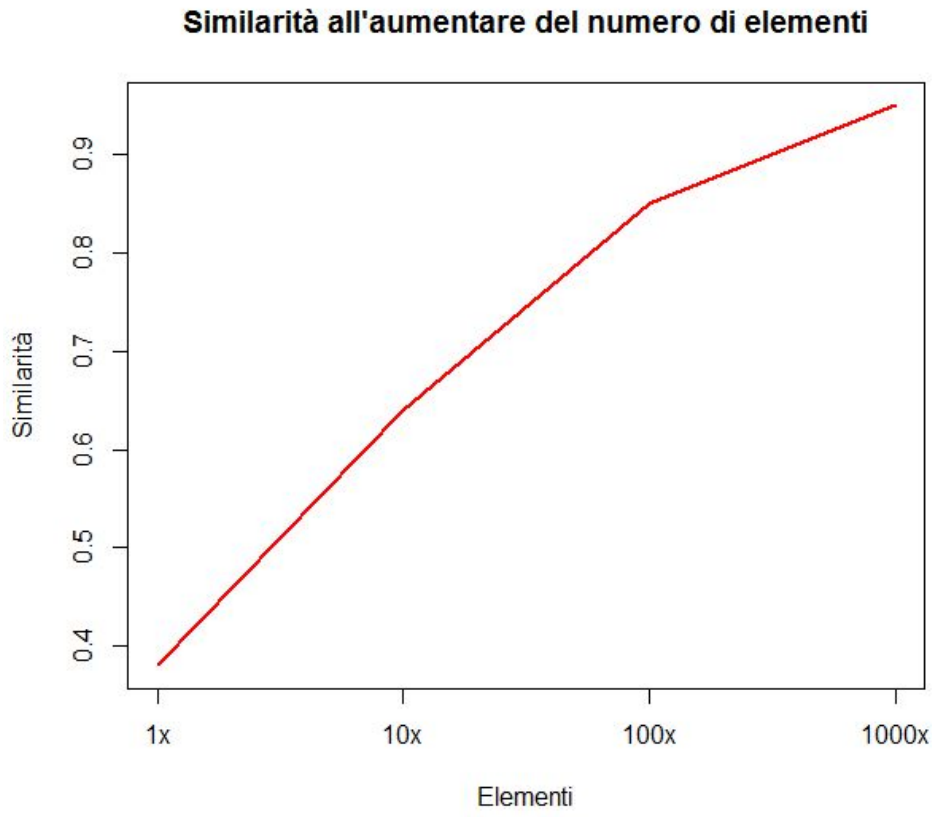
SON	2437	2735	***9710***
Toivonen	243	2613	17060

5.3 - Valutazione algoritmo Sampling

Per l'esecuzione dell'algoritmo Sampling, è stato utilizzato il file *chess.dat*, i cui elementi sono stati replicati 10, 100 e 1000 volte, e successivamente randomizzati per righe. E' stato quindi eseguito l'algoritmo 100 volte su ognuno di questi file, in modo da escludere eventuali risultati fortunati / sfortunati (che possono capitare in funzione della scelta delle transazioni, casuali, che effettua tale algoritmo).
Per la valutazione è stata utilizzata la distanza di Jaccard, definita tra due insiemi C_1 e C_2

$J(C_1, C_2) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_1|}$

E' stato quindi creato un grafico che rappresenta la similarità tra l'output dell'algoritmo Apriori e l'output del Sampling, all'aumentare del numero di candidati :



6 - Conclusioni

L'**Apriori** è un buon algoritmo con tempi di esecuzione accettabili. Ha però dei problemi di scalabilità all'aumentare del numero di campioni.
PCY funziona meglio di Apriori quando C2 con i relativi count per ogni coppia è una tabella troppo grande per entrare nella memoria principale; mentre il numero dei bucket frequenti in PCY è sufficientemente piccolo da entrare nelle memoria principale.
PCY funziona ancora meglio quando ci sono pochi bucket frequenti e questo accade quando la maggior parte delle coppie e cosi' poco frequente che le somme delle occorrenze nei bucket comuni (con lo stesso valore *hash*) non supera la soglia.
L'algoritmo **Sampling** utilizza una **probabilità fissata** (0.10 nel nostro caso) per decidere se una transazione deve essere considerata nel sottoinsieme o no. Possiamo notare come siano presenti dei *falsi positivi* e *falsi negativi*. I risultati sono comunque buoni.
L'algoritmo **SON** è stato implementato usando i *threads* su una macchina con 4 core, quindi i risultati potrebbero essere peggiori rispetto a quelli possibili da un'implementazione distribuita su macchine differenti o da una macchina che presenta un maggior numero di core (8 o 12 negli i7 si sesta e settima generazione).
Il numero di chunk scelti è proporzionale al numero di core (4).

Riferimenti Bibliografici

- **Park,Chen,Yu,** An effective hash-based algorithm for mining association rules, SIGMOD 95
- **Savasere**, **Omiecinski**, **Navathe**, An efficient Algorithm for Mining Association Rules in Large Databases,VLDB 95
- **Jure Leskovec**, **Anand Rajaraman**, **Jeff Ullman**, Mining of Massive Datasets