Project Work in Data Mining
Studio di Algoritmi di Frequent Itemeset Mining

Gregorio Piqué
gregorio.pique@edu.unifi.it

December 1, 2024

1 Introduzione

Il seguente documento costituisce il resoconto relativo al project work del corso di *Data Mining* del curriculum di Laurea Magistrale in Intelligenza Artificiale presso l'Università degli Studi di Firenze. Il lavoro ha richiesto uno studio approfondito e l'implementazione pratica di alcuni argomenti presentati durante il corso, in particolare di metodi di *Frequent Itemset Mining*.

2 Frequent Itemset Mining

Il *Frequent Itemset Mining* (da qui in poi *FIM*) è una tecnica di *data mining* fondamentale per identificare gruppi di oggetti frequentemente associati all'interno di grandi insiemi di dati, permettendo di scoprire pattern utili e inaspettati. L'obiettivo del FIM è estrarre nuove informazioni e conoscenza tramite metodi di analisi dati efficienti, rendendolo uno strumento chiave per comprendere meglio i comportamenti e le relazioni all'interno di dataset di grandi dimensioni.

Una delle principali applicazioni del FIM è l'estrazione di *association rules* (regole associative) che descrivono relazioni di tipo "*if... then*" tra gruppi di oggetti. Espresse nella forma $X \Rightarrow Y$, queste regole indicano una relazione tra due insiemi di oggetti disgiunti, come nella regola $\{onions, potatoes\} \Rightarrow \{burger\}$, che suggerisce la probabilità che un cliente di un supermercato che acquista cipolle e patate compri anche carne per hamburger. Le regole associative sono valutate tramite metriche come *supporto* e *confidenza*, fondamentali per quantificarne la rilevanza e l'affidabilità all'interno dei dati[1].

Il FIM e le *association rules* trovano ampio impiego in contesti strategici, dove permettono di individuare correlazioni tra prodotti o comportamenti dei clienti. Ad esempio, nel commercio e nella vendita, queste tecniche supportano l'ottimizzazione delle strategie di marketing e promozionali. Inoltre, risultano fondamentali per sistemi di raccomandazione personalizzati che suggeriscono prodotti basandosi sugli acquisti precedenti o su comportamenti simili di altri utenti. Oltre all'ambito commerciale, le *association rules* sono applicabili in settori come l'analisi del comportamento utente, la prevenzione delle frodi e l'analisi linguistica, individuando termini frequentemente associati nei contenuti online. Sono anche utilizzate per il controllo del plagio, evidenziando frasi ricorrenti tra documenti. In ciascuno di questi contesti, l'obiettivo è rilevare pattern utili a sostenere decisioni più informate e strategie mirate.

Il processo di FIM si articola in due fasi principali: identificazione dei *frequent itemset* e generazione delle regole di associazione. La fase più complessa e computazionalmente onerosa è la prima, su cui il progetto si è concentrato implementando in *Python* vari algoritmi per individuare i frequent itemset. Gli algoritmi spaziano per complessità ed efficienza, includendo il metodo *Brute-Force*, il *Counting Method*, l'algoritmo *Apriori* e alcune sue varianti come *PCY* e *MultiHash*; questi sono stati testati per confrontarne le prestazioni in termini di risultati e uso delle risorse.

3 Algoritmi di FIM

In questa sezione, si approfondiscono gli algoritmi implementati, evidenziando il loro funzionamento attraverso due passaggi fondamentali: la generazione dei candidati e il *support counting*, ovvero il

conteggio delle occorrenze effettive di un itemset candidato. La generazione dei candidati può variare da approcci naive, che considerano tutte le possibili combinazioni di item, a metodi più efficienti che garantiscono un insieme di candidati completo e privo di ridondanze, focalizzandosi solo su quelli rilevanti. Il support counting, a sua volta, può essere effettuato mediante approcci semplici che verificano la presenza di ciascun candidato in tutte le transazioni o tramite strutture dati come gli *hash tree*. Alcuni algoritmi, come Apriori e le sue varianti, sfruttano la proprietà di anti-monotonicità attraverso fasi intermedie come il *candidate pruning*, riducendo così il numero di candidati durante il support counting. Gli algoritmi implementati sono: *Brute-Force*, *Counting Method*, *Apriori*, *PCY*, *MultiHash*.

3.1 Brute-Force

Il primo algoritmo implementato è quello basato su un approccio di forza bruta: rappresenta il metodo di più semplice implementazione e comprensione, ma è in assoluto il meno performante sia in termini di tempo di esecuzione che di risorse necessarie al suo completamento.

Per la fase di identificazione dei candidati, il metodo si basa sulla generazione delle combinazioni di dimensione k a partire dall'intero insieme di N item. Consiste quindi nel calcolare $\binom{N}{k} = \frac{N!}{k!(N-k)!}$ itemset, approccio che ha una complessità di $O(N^k)$. Nell'implementazione Python, è stata usata la funzione *itertools.combinations()*[1]. Durante il *support counting*, viene confrontato ogni candiato con ciascuna transazione; se l'itemset considerato è contenuto nel basket, viene aggiornato il contatore per il supporto del candidato. La complessità è nell'ordine di $O(N \cdot M)$ con N il numero di transazioni e M il numero totale di candidati. Infine vengono ritornati come risultati finali tutti i candidati il cui support count è maggiore del valore soglia definito, per esempio gli itemset che sono presenti in almento 1% delle transazioni.

3.2 Counting Method

Il "metodo del conteggio" rappresenta il secondo metodo implementato; consiste nella generazione di tutti gli itemset di dimensione k presenti nelle transazioni considerate e nel conteggio di questi. La generazione dei candidati avviene basket per basket, da ciascuno dei quali vengono estratti tutti i sottoinsiemi di k elementi. Rispetto al metodo *brute-force*, questo approccio riduce la complessità a $O(N_i^k)$, dove N_i è pari alla dimensione dell'i-esimo basket[2]. Dopo la generazione, avviene il support counting, che consiste nel contare i duplicati dei candidati generati. Il passaggio finale di identificazione degli itemset frequenti, ovvero quelli il cui supporto è superiore alla soglia stabilita, è identico al metodo brute force.

3.3 Apriori

I metodi mostrati finora sono molto semplici, ma falliscono rapidamente come approcci quando il numero di itemset (ad esempio, le coppie) diventa molto grande. In casi come questi sorge il rischio di non avere sufficiente memoria per contarle tutte. L'algoritmo Apriori cerca di ridurre il numero di coppie da conteggiare, a costo di effettuare più passaggi sull'intero dataset[1]; in particolare, porta a leggere tutti i dati due volte per trovare, per esempio, le coppie frequenti. Il processo poi può essere esteso e generalizzato per l'identificazione di k-itemset.

L'idea di base di Apriori sta nella proprietà di *anti-monotonicità*, secondo la quale, se un oggetto i non compare in almeno s basket, nessuna coppia (o, in generale, nessun k-itemset) contenente l'oggetto i sarà presente in almeno s transazioni. In altre parole, un k-itemset è frequente solo se lo sono anche tutti i suoi sottoinsiemi[1, 3]. Grazie a questa proprietà, è tipicamente possibile rimuovere una grande quantità di itemset dall'insieme di candidati e, quindi, evitare di considerarli nella fasi successive, così da alleggerire, per esempio, quella di support counting.

3.3.1 Coppie Frequenti con Apriori

Le seguenti sotto-sezioni mostrano più in dettaglio il procedimento di Apriori per il caso di coppie frequenti. Il processo è anche rappresentato schematicamente nella Figura 1.

[1] https://docs.python.org/3/library/itertools.html

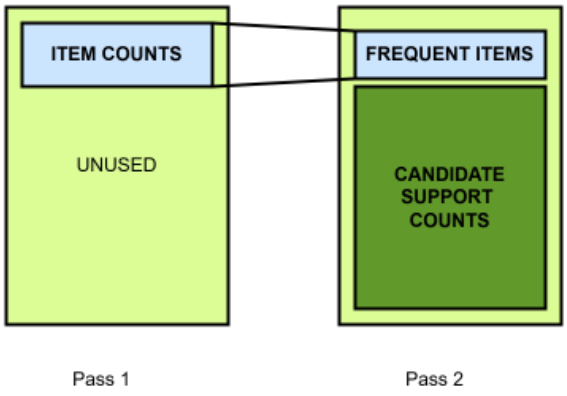

Figure 1: Passaggi di Apriori.

Passaggio 1
Nel primo passaggio di Apriori, l'algoritmo legge tutte le transazioni una per una e conta in memoria il numero di occorrenze di ciascun elemento che le costituisce. In particolare, quando viene letto un item, il valore di un contatore associato a quell'elemento viene incrementato. La memoria richiesta per questo primo passaggio di Apriori è proporzionale al numero totale di item presenti[2].

Passaggio Intermedio
Tra il primo e il secondo passaggio, avviene un'operazione intermedia che consiste nel filtrare gli oggetti in base alle loro occorrenze per individuare quelli il cui contatore ha un valore superiore alla soglia s stabilita: questi rappresentano gli item singoli frequenti e appartengono all'insieme chiamato F_1. La soglia s può essere scelta, ad esempio, come un valore pari all'1% del numero totale di basket. Lavorando con un numero sufficientemente grande di transazioni, scegliere una soglia abbastanza alta permette di ridurre significativamente la quantità di oggetti da considerare[1].

Passaggio 2
Nel secondo passaggio, Apriori riscansiona l'intero dataset per identificare i candidati composti da coppie di item. Questi candidati sono formati esclusivamente da coppie di item che, nel passaggio precedente, sono stati individuati come frequenti (cioè, il cui supporto supera la soglia minima s). In questo modo, si contano le occorrenze di itemset di dimensione 2 formati da elementi già noti come frequenti. Questa strategia è giustificata dalla proprietà di antimonotonicità: una coppia non può essere frequente se non lo sono anche i singoli elementi che la compongono. Grazie a questo metodo, Apriori evita di considerare coppie di item che non hanno alcuna possibilità di essere frequenti, ottimizzando così il processo di ricerca. Questa tecnica assicura che nessuna coppia potenzialmente frequente venga omessa[1]. La quantità di memoria richiesta per questo secondo passaggio è proporzionale al quadrato del numero di item frequenti F_1[2].

3.3.2 n-itemset Frequenti con Apriori

Come anticipato, il metodo Apriori può essere generalizzato per il caso di ricerca di itemset frequenti di dimensione arbitraria n: questo ripetendo n volte i passaggi precedentemente mostrati. Ad ogni iterazione k, l'algoritmo genera due insiemi costituiti da itemset di dimensione k: l'insieme di candidati C_k e quello di itemset frequenti F_k, come mostrato in Figura 2.

Ad ogni iterazione k, viene generato l'insieme di itemset candidati C_k utilizzando l'insieme F_{k-1}, ovvero degli elementi trovati essere frequenti all'iterazione precedente $k-1$. Generati i candidati, questi vengono filtrati tramite l'operazione di *support based pruning* che, sfruttando la proprietà di anti-monotonicità, permette di rimuovere (generalmente) una notevole quantità di itemset non frequenti, ignorandoli così nel successivo support counting. Vengono infine contate le occorrenze degli itemset rimanenti, grazie alle quali è possibile ricavare il nuovo insieme di elementi frequenti F_k. L'algoritmo termina se viene raggiunto il target della dimensione k degli itemset, o se F_k diventa vuoto (per monotonicità non potranno esistere itemset frequenti di più grande dimensione)[1].

Figure 2: k-itemset frequenti con Apriori.

3.3.3 Candidate Generation

Il primo passo per un'iterazione k consiste nella generazione degli itemset candidati. Come anticipato, l'insieme degli itemset candidati dovrebbe idealmente soddisfare alcune proprietà fondamentali. Dovrebbe essere completo, ovvero non trascurare nessun potenziale itemset frequente; privo di ridondanze, cioè non contenere duplicati; e dovrebbe considerare solo gli itemset rilevanti, ossia quelli che hanno una reale possibilità di essere frequenti. Seguono i metodi implementati.

Brute Force
Il primo metodo di generazione dei candidati utilizzabile con Apriori è il Brute Force, lo stesso impiegato nel metodo omonimo presentato nella Sezione 3.1. È un metodo molto semplice, ma che porta a individuare un numero generalmente molto elevato di candidati, rendendo molto costose le operazioni successive di *candidate pruning* e di *support counting*.

$F_{k-1} \cdot F_1$
Una prima alternativa è il metodo $F_{k-1} \cdot F_1$, che consiste nell'estendere ogni itemset frequente di dimensione $k-1$ (in F_{k-1}) con singoli elementi frequenti appartenenti a F_1, che non fanno parte del $(k-1)$-itemset considerato. Il metodo è completo, poiché ogni possibile k-itemset frequente è costituito da un itemset $k-1$ frequente e da un singolo elemento in F_1. Pertanto, il processo genererà come candidati tutti gli eventuali k-itemset frequenti. Nonostante il numero di candidati generati con questo approccio è notevolmente inferiore rispetto al metodo brute-force, il metodo $F_{k-1} \cdot F_1$ può comunque produrre un numero elevato di candidati, molti dei quali potrebbero non essere frequenti [2, 3]. Senza accorgimenti adeguati, il metodo non garantisce di evitare la generazione di duplicati. Questo può comunque essere evitato mantenendo un ordinamento lessicografico degli oggetti negli itemset frequenti. In questo modo, la generazione di un candidato sarebbe consentita solo se l'elemento singolo è "lessiconograficamente maggiore" dell'itemset che andrebbe ad estendere [1, 2, 3]. Ad esempio, dato l'itemset $\{onions, potatoes\}$, non è possibile aggiungere l'elemento $\{burger\}$ poiché "burger" precede lessicograficamente sia "onions" che "potatoes". Al contrario, nel caso di $\{burger, onions\}$, l'elemento *potatoes* potrebbe estendere l'itemset, generando così il 3-itemset $\{burger, onions, potatoes\}$, poiché l'ordine viene rispettato correttamente.

$F_{k-1} \cdot F_{k-1}$
L'ultimo metodo implementato per la generazione dei candidati è $F_{k-1} \cdot F_{k-1}$; può essere considerato un'evoluzione del precedente $F_{k-1} \cdot F_1$, ed è il metodo standard utilizzato in Apriori. Questo processo consiste nell'estendere un $(k-1)$-itemset combinando due itemset A e B di F_{k-1}, a condizione che, ordinati lessicograficamente, abbiano un prefisso di $k-2$ elementi uguali, e che l'ultimo elemento di A sia diverso dall'ultimo di B. Il candidato generato sarà quindi composto dai primi $k-2$ elementi comuni a entrambi gli itemset, seguiti da a_{k-1} e b_{k-1}, ordinati [2, 3]. Dati gli itemset $A = \{burger, onions, potatoes\}$ e $B = \{burger, onions, tomatoes\}$, questi sono combinabili tra loro con $F_{k-1} \cdot F_{k-1}$ per generare un candidato di dimensione $k = 4$, perché (i) costituiti da $k-2 = 2$ elementi uguali (cioè $\{burger, onions\}$) e (ii) a_{k-1} e b_{k-1} sono diversi ($\{potatoes\}$ e $\{tomatoes\}$). L'itemset candidato generato sarà quindi $C = \{burger, onions, potatoes, tomatoes\}$.

3.3.4 Candidate Pruning

Una volta generati i candidati, Apriori applica una fase di *pruning* per eliminare quelli che sicuramente non sono frequenti, sfruttando la proprietà antimonotonica. In pratica, se un candidato di dimensione k contiene almeno un sottoinsieme di dimensione inferiore a k che non è stato trovato frequente nelle iterazioni precedenti, allora anche il candidato stesso non può essere frequente e viene quindi scartato. Grazie alla proprietà di antimonotonicità, non è necessario verificare esplicitamente che

tutti i sottoinsiemi di dimensioni minori di $k-1$ siano frequenti. Apriori genera candidati estendendo esclusivamente gli itemset di F_{k-1}, garantendo così che tutti i sottoinsiemi di dimensione inferiore siano già frequenti [3]. Inoltre, l'operazione di pruning cambia leggermente a seconda del metodo utilizzato per la generazione dei candidati. Se è stato utilizzato il metodo Brute Force, è necessario controllare tutti i k-sottoinsiemi di dimensione $k-1$; se è stato utilizzato $F_{k-1} \cdot F_1$, sono $k-1$ gli itemset da verificare; se invece è stato utilizzato $F_{k-1} \cdot F_{k-1}$, gli itemset da controllare sono $k-2$ [2, 3].

3.3.5 Support Counting

L'ultimo passaggio di Apriori è il *support counting*, operazione che consiste nel contare le occorrenze dei candidati rimasti dopo il *pruning*. Anche in questo caso vi sono diversi approcci utilizzabili, tra i quali un metodo Brute Force e uno che utilizza una struttura dati ad albero.

Brute Force
Il metodo Brute Force consiste nel confrontare tutti i candidati con ogni transazione, aggiornando i contatori delle occorrenze degli itemset presenti nella transazione. Questo metodo è semplice ma al crescere del numero di transazioni o di candidati diventa computazionalmente molto costoso [3].

Support Counting con Hash Tree
Un metodo alternativo consiste nel costruire e poi sfruttare un *hash tree*: Apriori partiziona i candidati in bucket e li memorizza in una struttura ad albero utilizzando una funzione di hash, che può avere una forma come $h(p) = (p-1) mod\ m$, dove m rappresenta la dimensione dell'itemset. Questo processo consente di distribuire un itemset p nel ramo corrispondente a $h(p)$, creando un albero in cui i candidati si trovano nei nodi foglia. Dopo la fase iniziale di costruzione dell'hash tree, viene svolto il support counting. In questa fase, l'albero viene percorso utilizzando gli itemset presenti in ciascuna transazione, mappando così questi ultimi nei rispettivi bucket. Questo approccio permette di confrontare gli itemset delle transazioni solo con quelli di un bucket specifico dell'albero, evitando di dover esaminare l'intero insieme di candidati[3].

3.4 PCY

La prima variante implementata dell'algoritmo Apriori è PCY, che introduce ottimizzazioni nella gestione delle risorse. PCY si concentra sull'uso efficiente della memoria, in particolare durante il primo passaggio di Apriori, dove spesso si ha una notevole quantità di memoria inutilizzata (vedi Figura 1) [1]. Sfruttando questo aspetto, PCY ottimizza il conteggio del supporto degli itemset. Nel primo passaggio, viene creata una hash table di dimensioni tali da occupare il massimo possibile di memoria. Durante questa fase, vengono contate le occorrenze dei singoli item e generate tutte le possibili coppie di itemset presenti nel basket, che vengono associate a un bucket della tabella tramite una funzione di hash, incrementando i relativi contatori.

Alla fine del primo passaggio, ogni bucket avrà un contatore corrispondente alla somma delle occorrenze delle coppie associate. Se il contatore di un bucket è maggiore o uguale alla soglia di supporto minimo s, quel bucket viene considerato frequente; in caso contrario, si può affermare con certezza che tutte le coppie ad esso associate non sono frequenti. La hash table viene quindi ridotta a una bitmap, in cui ogni elemento corrisponde a un bucket: un valore di 1 indica un bucket frequente, mentre 0 rappresenta un bucket non frequente. Questa bitmap viene utilizzata nel secondo passaggio di PCY durante il support counting, permettendo di ridurre il numero di candidati considerati: un itemset sarà ritenuto valido se composto da singoli item frequenti e mappato su un bucket frequente della bitmap [1].

3.5 Multihash

La seconda variante di Apriori implementata è Multihash, che estende l'approccio di PCY utilizzando più tabelle hash di dimensioni minori, ognuna associata a una funzione di hash indipendente. Come in PCY, nel primo passaggio vengono contati gli item singoli e generate le combinazioni di coppie di item, che vengono poi mappate su ciascuna delle n tabelle hash usando le rispettive funzioni. I contatori dei bucket corrispondenti in ogni tabella vengono incrementati, e, al termine del passaggio, ciascuna tabella viene ridotta a una bitmap. Nel secondo passaggio di support counting, un itemset è

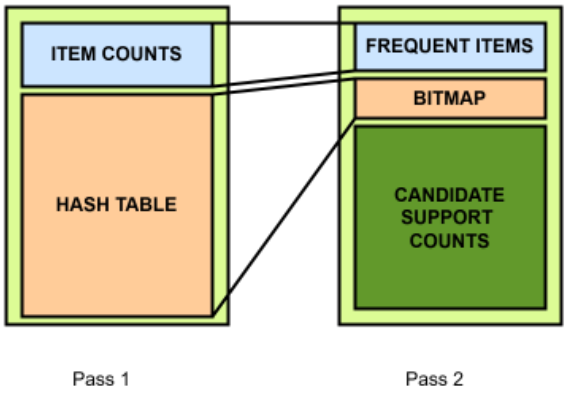

Figure 3: Passaggi di PCY.

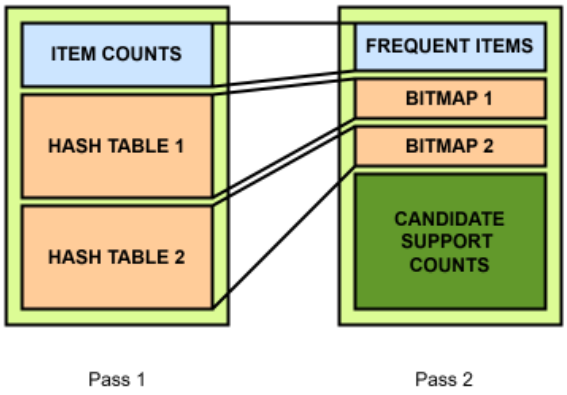

Figure 4: Passaggi di Multihash.

considerato valido solo se soddisfa le stesse condizioni di PCY, con la differenza che ora il mappaggio avviene su tutte le n bitmap: l'itemset deve corrispondere a bucket frequenti in ogni bitmap. Se anche un solo bucket contiene uno 0, l'itemset viene scartato, riducendo il numero di candidati da verificare [1, 2]. Questo processo è rappresentato nella Figura 4.

4 Implementazione

Il progetto ha richiesto l'implementazione di codice Python per la risoluzione del problema di FIM con i diversi algoritmi presentati nelle sezioni precedenti. Le classi implementate e le relazioni tra queste sono mostrate nelle seguenti sottosezioni. Nella Appendice A è riportanto un caso d'uso del progetto implementato. Il codice sorgente è accessibile al repository *Git* `https://github.com/Pikerozzo/dm-frequent-itemsets`.

4.1 Algoritmi

Le classi utilizzate per rappresentare gli algoritmi sono illustrate nella Figura 5. Gli algoritmi *BruteForce*, *CountingMethod*, *Apriori*, *PCY* e *MultiHash* sono rappresentati ciascuno da una classe, che eredita dalla classe astratta *Algorithm*. Quest'ultima definisce il metodo comune *find_frequent_itemsets()*, poi implementato in modo specifico nelle varie sottoclassi.

4.1.1 BruteForce

BruteForce è il più semplice tra gli algoritmi implementati. L'unico metodo che possiede è quello ereditato dalla classe *Algorithm*, mostrato nell'algoritmo 1.

Algorithm 1 BruteForce

Input: transactions, itemset_size, min_sup

all_items ← items *in* transactions

candidates ← combinations(*all_items*, *itemset_size*)

support_counts ← {}
for ***each*** *transaction* **do**
  for ***each*** *candidate* **do**
    if *candidate* ⊆ *transaction* **then**
      support_counts[candidate] ← support_counts[candidate] + 1
    end
  end
end

frequent_itemsets ← [ ]
for ***each*** *(itemset, count)* ***in*** *support_counts* **do**
  if *count* ≥ *min_sup* **then**
    frequent_itemsets.append([itemset, count])
  end
end
Output: frequent_itemsets

4.1.2 CountingMethod

Il secondo algoritmo implementato è *CountingMethod*. Anche questo presenta un unico metodo, illustrato nell'algoritmo 2. Il passaggio finale, che identifica gli itemset frequenti a partire dai *support_counts*, è identico a quello del metodo *BruteForce*; per questo motivo il passaggio è stato ridotto e semplificato. Si noti che il metodo considera tutte le possibili coppie di item basket per basket, ma queste non vengono mantenute tutte in memoria contemporaneamente: per ciascuna transazione, vengono generati i candidati e subito ne vengono incrementati i rispettivi contatori (per ulteriori dettagli, si rimanda all'implementazione [4])

Algorithm 2 CountingMethod

Input: transactions, itemset_size, min_sup

candidates ← [ ]

support_counts ← {}
for ***each*** *transaction* **do**
  candidates ← combinations(*transaction*, *itemset_size*)
  for ***each*** *candidate* **do**
    support_counts[candidate] ← support_counts[candidate] + 1
  end
end

frequent_itemsets ← itemsets with support ≥ min_sup

Output: frequent_itemsets

4.1.3 Apriori

Il terzo algoritmo è *Apriori*, che include una serie di metodi di supporto per le diverse fasi (generazione, pruning e conteggio dei candidati), ciascuno con le sue varianti implementate (vedi Sezione 3.3).

Algorithm 3 Apriori

Input: transactions, itemset_size, min_sup

$k = 1$

$F_k = \{item | item \in I \cap \sigma(\{item\}) \geq N \times minsup\}$

repeat

  $k = k + 1$

  $C_k = generate\_candidates(F_{k-1})$

  $C_k = prune\_candidates(C_k, F_{k-1})$

  $hash\_tree = HashTree.insert\_candidates(C_k)$

  $hash\_tree.count\_supports(transactions)$

  $F_k = hash\_tree.get\_itemsets\_with\_support(min\_sup)$

until $F_k = \emptyset \cup k = itemset\_size$;

Output: F_k

4.1.4 PCY

Il quarto algoritmo è *PCY*, la prima variante di Apriori. Questo introduce una bitmap come campo di classe, utilizzata per ridurre il numero di candidati durante il support counting (vedi Sezione 3.4). L'algoritmo riutilizza alcuni metodi ereditati dalla classe *Apriori* e ne reimplementa altri, come l'identificazione di F_1, la generazione del set C_k e il support counting, durante i quali la bitmap viene definita, costruita e utilizzata.

Algorithm 4 PCY

Input: transactions, itemset_size, min_sup

$k = 1$

$F_k,\ bitmap = extract\_F_1\_and\_build\_bitmap(transactions)$

repeat

  $k = k + 1$

  $C_k = generate\_candidate\_with\_bitmap(F_{k-1}, bitmap)$

  $C_k = prune\_candidate(C_k, F_{k-1})$

  $hash\_tree = HashTree.insert\_candidates(C_k)$

  $hash\_tree.count\_supports(transactions)$

  $bitmap = build\_bitmap()$

  $F_k = hash\_tree.get\_itemsets\_with\_support(min\_sup)$

until $F_k = \emptyset \cup k = itemset\_size$;

Output: F_k

4.1.5 MultiHash

L'ultimo algoritmo è *MultiHash*, la seconda variante di Apriori. Questa classe estende *PCY*, ereditandone tutti i metodi (vedi Sezione 3.5). Il funzionamento di MultiHash è identico a quello di PCY, con la sola differenza che utilizza un numero maggiore di bitmap, ciascuna di dimensioni ridotte.

4.2 HashTree

La classe *HashTree* fornisce le funzionalità necessarie per costruire un albero che gestisce e conta le occorrenze degli itemset candidati, con i nodi rappresentati e costituiti da oggetti della classe *HashTreeNode*. La costruzione dell'albero avviene attraverso il metodo *insert_candidates*(), che utilizza una funzione di hash per distribuire gli itemset nei rami e inserire gli itemset candidati nei nodi foglia. Se un nodo supera il numero massimo di itemset consentiti, viene suddiviso in sotto-nodi tramite la funzione *split_node*(). Il conteggio dei supporti è gestito dal metodo *count_supports*(), che attraversa l'albero utilizzando la stessa funzione di hash impiegata nella costruzione. Quando si raggiunge una foglia contenente un candidato corrispondente all'itemset considerato, il suo contatore delle occorrenze viene

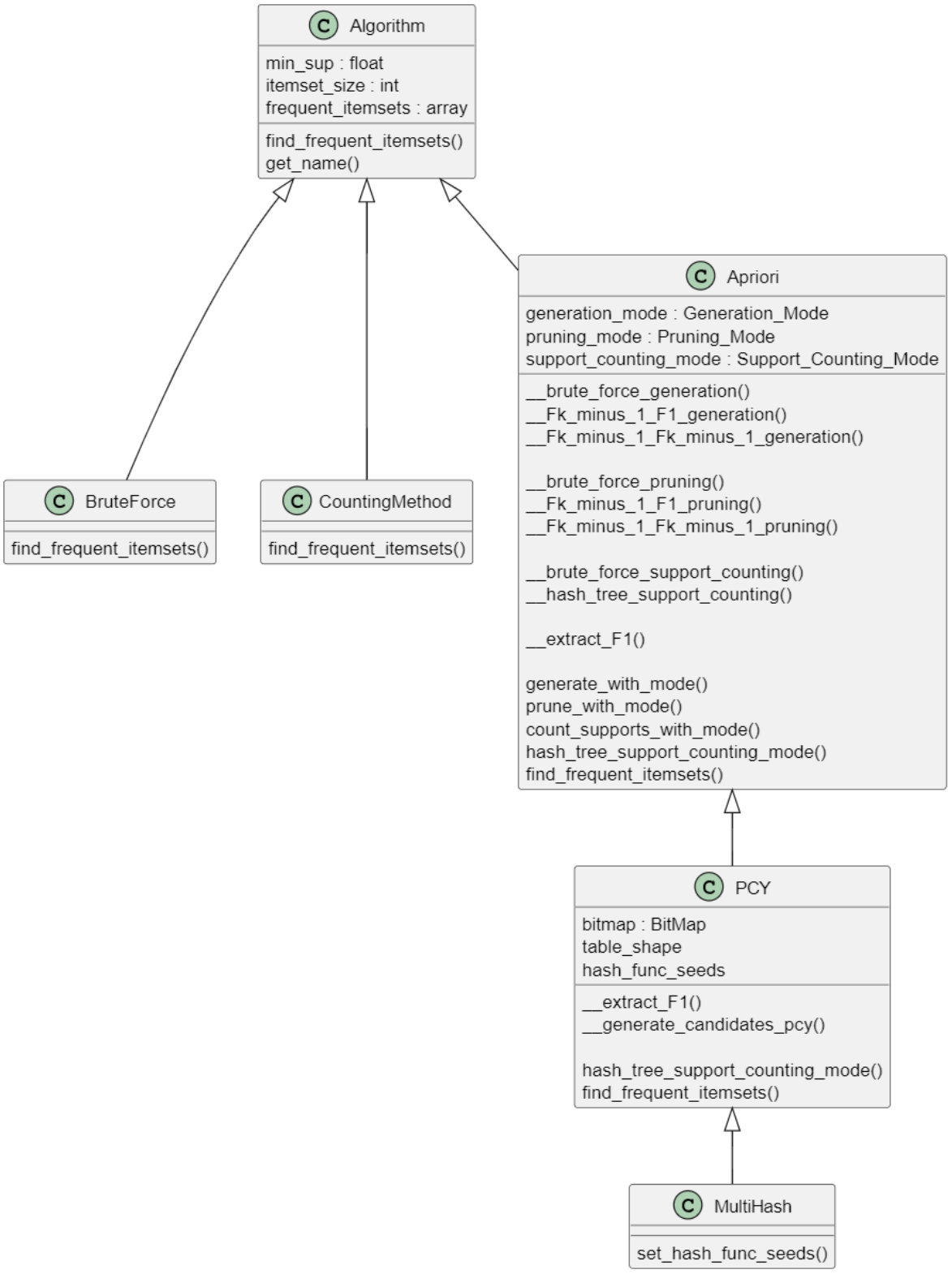

Figure 5: Classi per algoritmi.

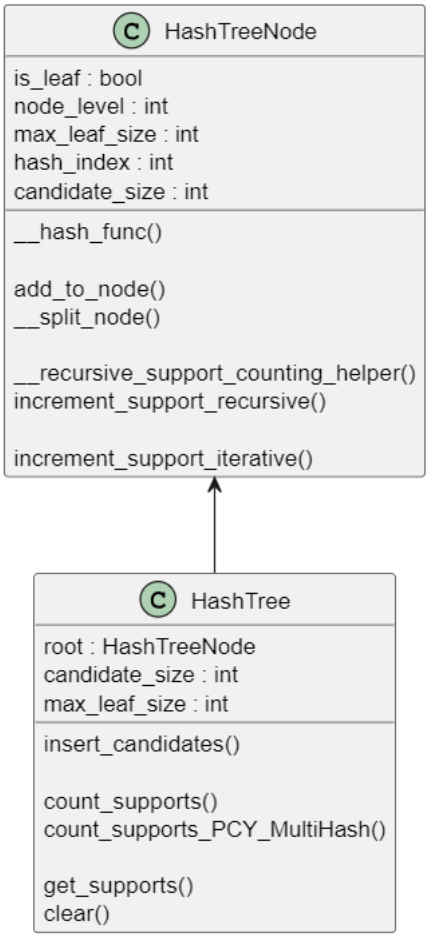

Figure 6: Classe per il *hash tree*.

incrementato. Esiste anche una variante di questo metodo, utilizzata dagli algoritmi PCY e MultiHash, che incrementa i contatori delle loro tabelle di hash per la costruzione delle bitmap. Infine, la classe include anche un metodo per recuperare i supporti degli itemset candidati. La struttura della classe è illustrata nella Figura 6.

4.3 BitMap

La classe *BitMap* mette a disposizione le funzionalità per gestire le bitmap usate dai metodi PCY e MutliHash. Il campo *bitmaps* è costituito da un array di oggetti di tipo *bitarray*[2], una struttura dati che permette di rappresentare in modo efficiente array booleani. La struttura della classe è mostrata nella Figura 7.

4.4 Utils

La classe *Utils* offre funzionalità utili per l'esecuzione generale degli algoritmi, tra cui la quantificazione delle dimensioni delle hashtables utilizzate in PCY e MultiHash, e una funzione per recuperare i nomi degli item a partire dai loro *ID* numerici. La struttura della classe è rappresentata nella Figura 8.

5 Risultati

In questa sezione, vengono analizzati e commentati i risultati ottenuti dai test condotti. Sono state esplorate diverse configurazioni del problema di FIM, utilizzando vari algoritmi e differenti numeri

[2]https://pypi.org/project/bitarray/

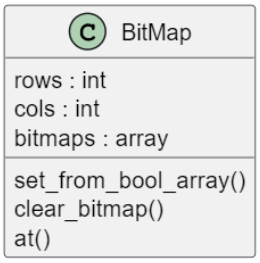

Figure 7: Classe per la bitmap.

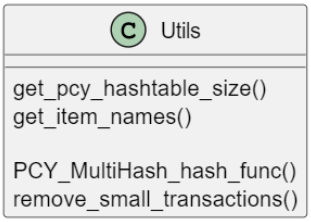

Figure 8: Classe con funzioni di supporto.

di transazioni. Durante i test, sono stati registrati i risultati, come il numero di candidati generati e il numero di itemset frequenti identificati, monitorando anche le risorse impiegate, quali tempo di esecuzione e utilizzo della memoria. I tempi di esecuzione sono stati misurati con librerie standard di Python, mentre il consumo di memoria è stato monitorato tramite la libreria *memory_profiler*[3]. I grafici seguenti presentano i risultati dei test, in cui sono state ricercate coppie frequenti con un supporto minimo fissato all'1%. I dati utilizzati per questi test provengono da un dataset *Kaggle* [5].

5.1 Tempo di esecuzione

La Figura 9 illustra il tempo di esecuzione totale dei vari algoritmi in relazione al numero di transazioni utilizzate. Si osservano tre comportamenti principali: quello del metodo *BruteForce*, del *Counting-Method* e quello di *Apriori* con le sue varianti. Come previsto, il metodo *BruteForce* presenta la scalabilità peggiore; anche con sole 100 transazioni, i suoi tempi di esecuzione superano di gran lunga quelli degli altri algoritmi. Con 1000 transazioni, i tempi oscillano tra i 30 e i 40 minuti, rendendo impraticabili ulteriori test con un numero maggiore di transazioni. L'algoritmo *Counting-Method* mostra inizialmente una scalabilità migliore rispetto al metodo *BruteForce*, ma a partire da circa $10k$ transazioni, i tempi di completamento incrementano notevolmente, richiedendo fino a dieci minuti per $1M$ transazioni. I metodi iterativi, che includono *Apriori* e le sue varianti, mantengono quasi costantemente un andamento simile, con *Apriori* che risulta essere il più veloce, seguito da *PCY* e *MultiHash*.

L'analisi del tempo di completamento segue nelle Figure 10 e 11, dove vengono illustrati i tempi necessari per i due passaggi di Apriori e delle sue varianti. Il primo passaggio riguarda l'estrazione di F_1, e per le varianti PCY e MultiHash include anche la costruzione della tabella di hash e l'incremento dei contatori dei suoi bucket. Il secondo passaggio si concentra sulla generazione, pruning e conteggio degli itemset candidati. Nella Figura 10, il tempo per il *pass 1* evidenzia che Apriori è il metodo più rapido tra i tre analizzati. Questo risultato è attribuibile alla sua semplicità, in quanto non richiede di identificare tutte le coppia di item in tutte le transazioni, mapparle su bucket nelle hashtable e incrementarne i contatori. Seguono a una certa distanza le varianti PCY e MultiHash, che mostrano prestazioni simili in tutte le dimensioni del dataset. La Figura 11 analizza il tempo necessario per il *pass 2*. In questa fase, le varianti iniziano rapidamente, probabilmente a causa del basso numero di transazioni che comporta la generazione 0 candidati. A partire da $500k$ transazioni, le varianti iniziano a mostrare tempi di completamento migliori (fino a 50 secondi più rapidi) rispetto ad Apriori, che deve gestire un numero significativamente maggiore di itemset candidati e eseguire il conteggio del supporto

[3] https://pypi.org/project/memory-profiler/

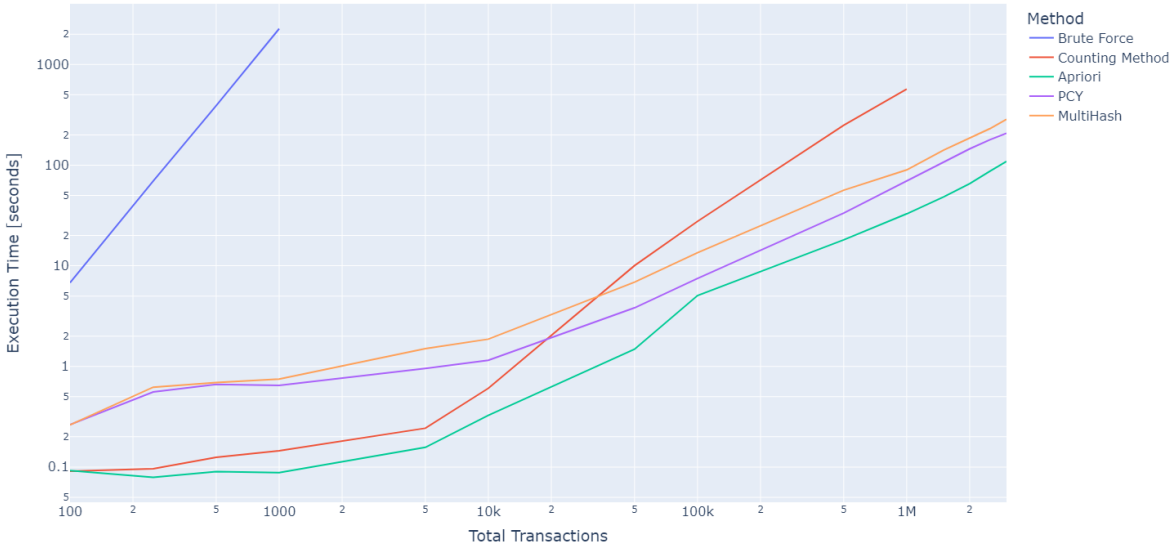

Figure 9: Tempi toale di esecuzione per numero di transazioni.

per tutti questi.

5.2 Candidati generati e itemset frequenti

Le figure che seguono analizzano il numero di itemset generati come candidati dai vari metodi (Figura 12) e gli effettivi itemset frequenti individuati (Figura 13), entrambi in relazione al numero di transazioni utilizzate.

Riguardo ai candidati generati, il metodo BruteForce produce tutte le possibili combinazioni di coppie di itemset a partire dall'intero insieme di N item. Già con $1k$ transazioni, il numero di candidati supera i $10M$. L'algoritmo CountingMethod, sebbene generi tutte le combinazioni, opera basket per basket, riducendo notevolmente la complessità e il numero di itemset generati, poiché considera solo quelli presenti nel dataset. Tuttavia, il numero di candidati cresce rapidamente fino a quasi $100M$ con $1M$ di transazioni. Apriori impiega tecniche più efficienti, come $F_{k-1} \cdot F_{k-1}$, mantenendo il numero di candidati costantemente più basso rispetto agli approcci di forza bruta. Le varianti di Apriori, inoltre, riducono ulteriormente il numero di candidati grazie all'uso di bitmap per l'identificazione dei bucket frequenti.

La Figura 13 mostra il numero di itemset frequenti identificati. Nonostante le significative differenze nel numero di candidati generati, tutti i metodi riescono a individuare lo stesso numero di itemset frequenti, stabilizzandosi intorno a 15 itemset a partire da circa $50k$ transazioni. Questo fenomeno potrebbe derivare dal fatto che, anche in dataset di grandi dimensioni, l'insieme degli itemset frequenti tende a rimanere quasi invariato, con gli itemset considerati frequenti con un numero inferiore di transazioni che risultano effettivamente tali anche con un dataset più ampio.

5.3 Memoria utilizzata

I grafici seguenti illustrano l'uso della memoria di ciascun metodo in relazione al numero di transazioni. La misurazione è stata effettuata utilizzando la funzione *memory_usage* della libreria *memory_profiler*, che registra regolarmente la quantità di memoria utilizzata a intervalli predefiniti.

La Figura 14 illustra l'uso della memoria con $1k$ transazioni. La misurazione è effettuata ogni 0.1 secondi per tutti i metodi, ad eccezione dell'algoritmo BruteForce, per il quale l'intervallo è fissato a 1 secondo a causa della sua lunga durata di esecuzione. Dal grafico si può notare che BruteForce richiede una quantità di memoria significativamente superiore rispetto agli altri metodi. Tra gli altri algoritmi, le differenze nell'uso della memoria non sono particolarmente marcate, probabilmente a causa del numero relativamente contenuto di transazioni, che non mette in evidenza gli effetti delle diverse

Figure 10: Tempi di esecuzione durante il Pass1 di Apriori e varianti, per numero di transazioni.

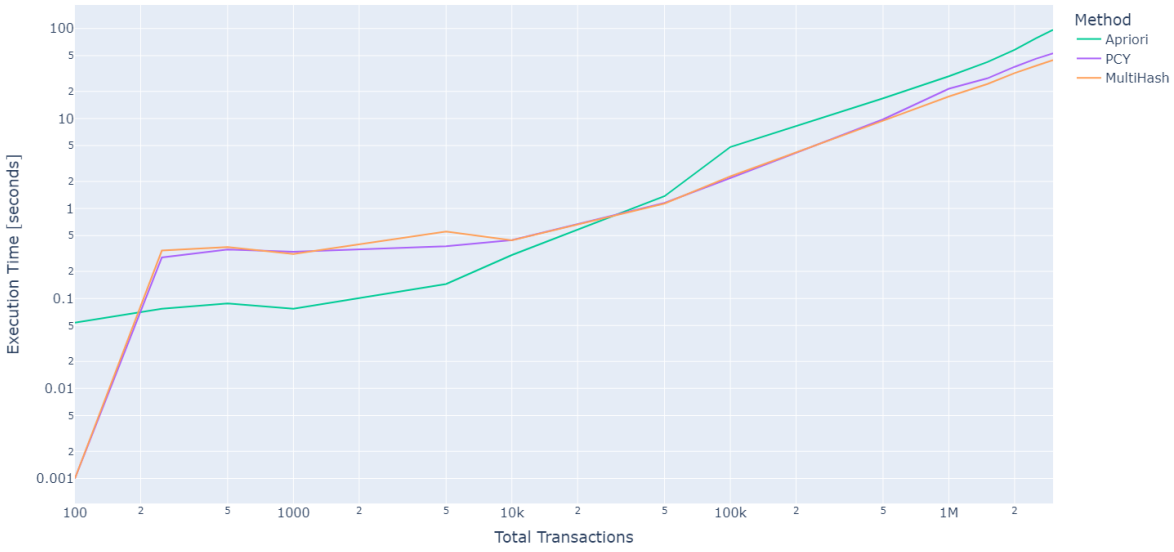

Figure 11: Tempi di esecuzione durante il Pass2 di Apriori e varianti, per numero di transazioni.

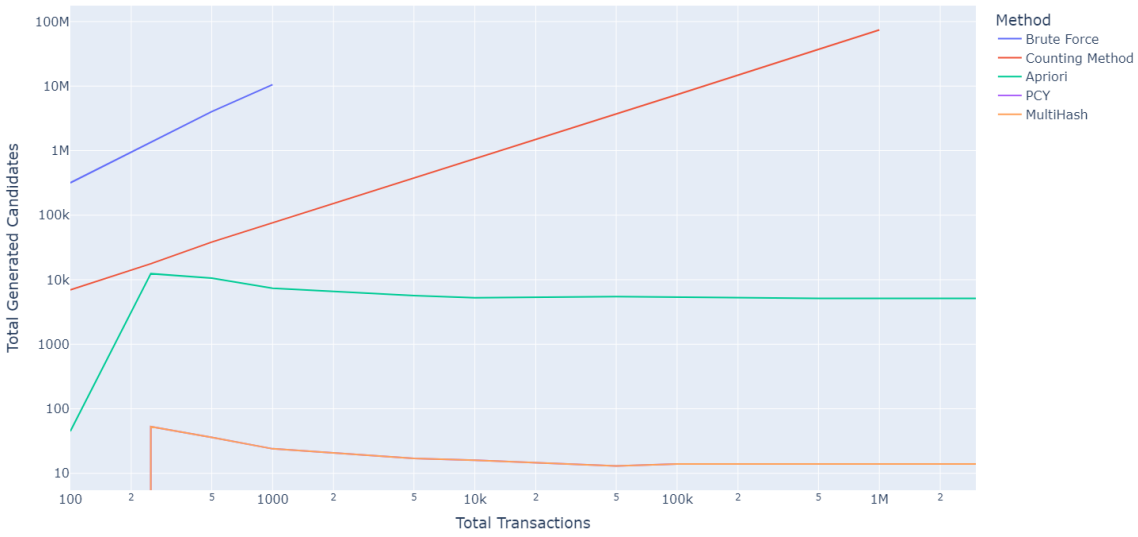

Figure 12: Numero di itemset candidati generati per numero di transazioni.

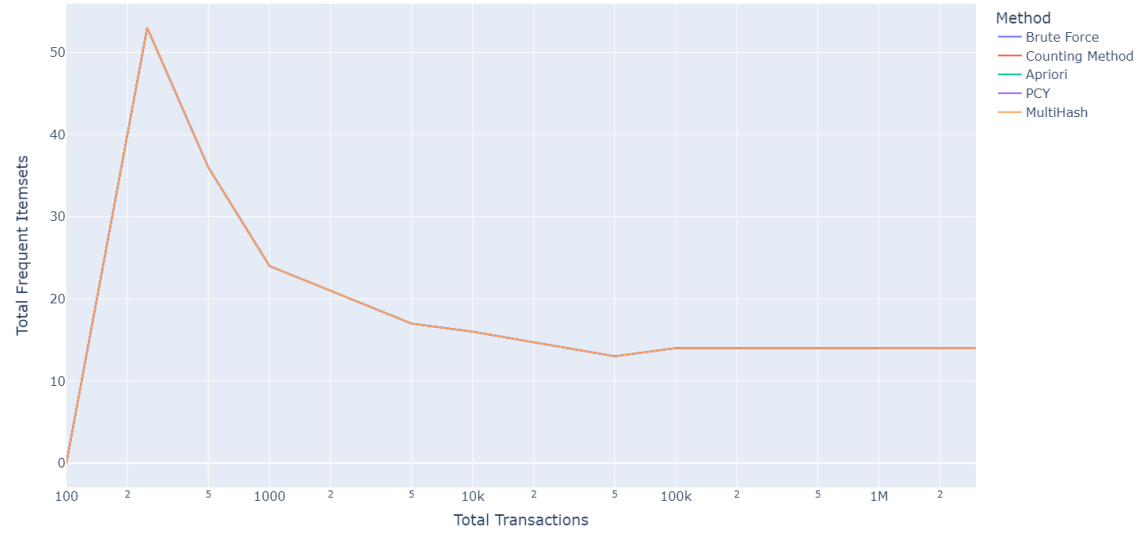

Figure 13: Numero di itemset frequenti per numero di transazioni.

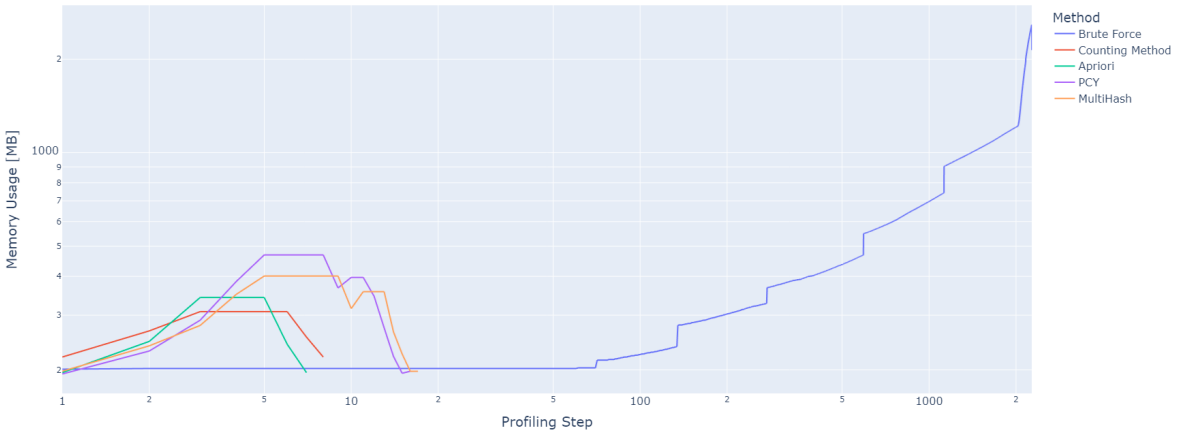

Figure 14: Memoria utilizzata con 1k transazioni.

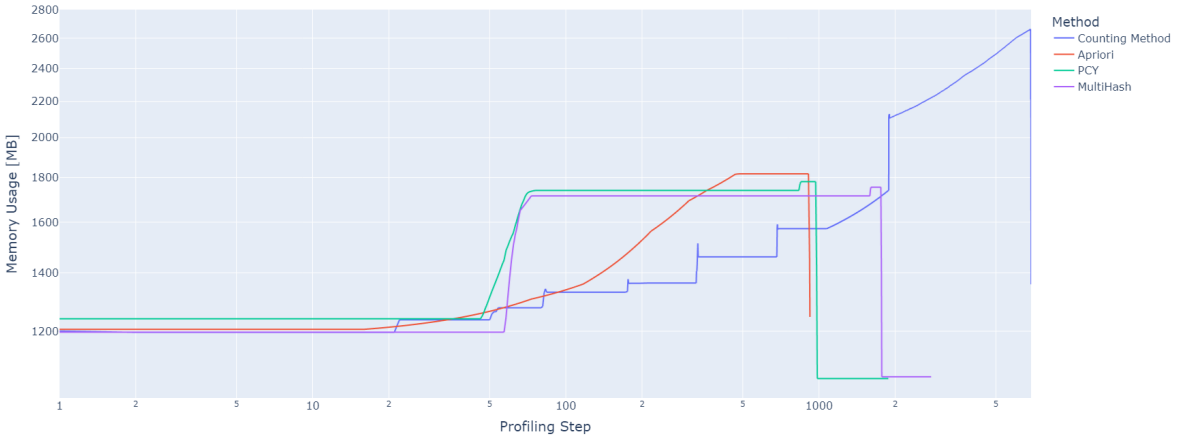

Figure 15: Memoria utilizzata con 1M transazioni.

tecniche adottate. Si notano comunque differenze tra gli andamenti dei metodi CountingMethod e Apriori con quelli di PCY e MultiHash.

La Figura 15 mostra invece l'uso della memoria con $1M$ di transazioni. Per motivi pratici, il metodo BruteForce è stato testato solo fino a $1k$ transazioni e pertanto non è incluso in questo grafico. Si osserva che l'algoritmo CountingMethod consuma progressivamente più memoria, arrivando a circa 2.7GB; anche Apriori mostra un aumento costante, arrivando a un massimo di oltre 1.8GB. In questo contesto, per le hash table di PCY e MultiHash è stata utilizzata una memoria totale di circa 500MB; nel caso di MultiHash, questa è suddivisa in due tabelle. Qui emergono i vantaggi delle varianti, che, pur richiedendo più tempo rispetto ad Apriori per completare l'analisi (soprattutto a causa del più lento *pass 1*), utilizzano una quantità minore di memoria, arrivando a un massimo di circa 1.7GB.

6 Conclusioni

In questo progetto sono stati studiati e implementati diversi algoritmi per il FIM, tra cui l'approccio *Brute Force*, il *Counting Method*, l'algoritmo *Apriori* e due sue varianti: *PCY* e *MultiHash*. L'analisi dei risultati ha evidenziato i punti di forza e le limitazioni di ciascun metodo, considerando aspetti quali la semplicità di implementazione, la velocità di esecuzione e la quantità di memoria richiesta. Questo lavoro potrebbe essere ampliato analizzando altri algoritmi ottimizzati per dataset di grandi dimensioni oppure proseguendo il processo con l'identificazione e l'estrazione delle *association rules*.

A Caso d'Uso

In questa sezione vengono illustrate le modalità di utilizzo delle funzionalità implementate. L'esempio seguente riguarda la ricerca di coppie frequenti all'interno di un dataset contenente $100k$ transazioni, utilizzando un supporto minimo del 1%. L'esempio utilizza l'algoritmo MultiHash, configurato con 2 bitmap di dimensione totale pari a 500MB.

```
from utils import *
from algorithms import *

# Load transaction dataset
transactions, items = load_data(dataset_size=100_000)

# Create the algorithm instance
alg = MultiHash(tot_megabytes=500, tot_hash_tables=2)

# Find frequent pairs
alg.find_frequent_itemsets(data=transactions, itemset_size=2, min_sup=0.01)

# Print some results
tot_freq_items = len(alg.frequent_itemsets)
print(f'Found {tot_freq_items} frequent itemsets with size {alg.itemset_size}')

# Print the top 5 frequent itemsets
alg.frequent_itemsets = sorted(alg.frequent_itemsets, key=lambda x:x[1], reverse=True)
itemsets_with_names = Utils.get_item_names(alg.frequent_itemsets, items)
for i in range(min(5, tot_freq_items)):
    print(f'{itemsets_with_names[i]} has support {alg.frequent_itemsets[i][1]}')
```

Segue l'output finale dell'esempio mostrato sopra, che riporta le cinque coppie più frequenti ritrovate:

```
Found 16 frequent itemsets with size 2
['Bag of Organic Bananas' 'Organic Strawberries'] has support 2353
['Bag of Organic Bananas' 'Organic Hass Avocado'] has support 1818
['Banana' 'Organic Avocado'] has support 1719
['Bag of Organic Bananas' 'Organic Baby Spinach'] has support 1709
['Banana' 'Large Lemon'] has support 1659
```

References

[1] J. Leskovec, A. Rajaraman, and J.D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2014.

[2] S. Marinai. *Data Mining*. Lecture slides. INM & IAM, Academic Year 2023/2024. University of Florence, IT.

[3] P.N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining (2nd Edition)*. Pearson, 2019. ISBN: 0321321367.

[4] G. Piqué. *Pikerozzo / dm-frequent-itemsets*. URL: https://github.com/Pikerozzo/dm-frequent-itemsets. (accessed: 09.11.2024).

[5] *Kaggle - Frequent Itemsets and Association Rules*. URL: https://www.kaggle.com/code/msp48731/frequent-itemsets-and-association-rules/input. (accessed: 20.08.2024).