UNIVERSITÀ
di VERONA

Ingegneria e Scienze Informatiche
Big Data

Algoritmo S.O.N. in PySpark

Leonardo Zecchin VR462541
25 agosto 2022

Consegna

In questo documento si vuole spiegare le scelte effettuate durante lo sviluppo del progetto valido per l'esame di Big Data.
La **consegna** è la seguente: Implementazione Map Reduce dell'algoritmo A-Priori per i Frequent itemset in particolare dell'algoritmo S.O.N.

Indice

1 Introduzione

1.1 Spiegazione Progetto

L'obiettivo del progetto è di implementare un algoritmo per la ricerca dei **Frequent Itemset** attraverso l'utilizzo del framework **Map Reduce** in **Spark** oppure **Hadoop**.
Il progetto chiede di implementare l'**algoritmo SON** con uno sviluppo normale nella propria macchina e uno sviluppo parallelo utilizzando per esempio Spark e confrontare i due risultati.

1.2 Frequent Itemset

Un **item** è un oggetto/elemento che fa parte di un insieme di oggetti che viene detto **itemset**; esso è un insieme che contiene uno o più item e fa parte di un **basket**.
Il problema consiste nel trovare gli itemset che sono **frequenti**, cioè gli item che appaiono nella maggior parte dello stesso basket.
Introduciamo il termine *supporto* che indica il numero di basket in cui I appare; I si dice *frequent* se il suo supporto è maggiore o uguale di s, *support threshold*.

1.3 MapReduce

MapReduce è un modello di programmazione per processare e generare grandi insiemi di dati in *cluster* con l'utilizzo di algoritmi paralleli e distribuiti.
Questo modello facilita l'elaborazione dei dati perché divise il file in *chunk*, cioè in partizioni che vengono processate in **parallelo** e in conclusione aggregate in un unico output.Questo è uno dei vantaggi ed inoltre un suo punto di forza sta nel fatto che la logica viene eseguita nello stesso punto in cui risiedono i dati.
MapReduce è contraddistinto da due fasi:

1. **Map**, prende l'input e applica la funzione di map alla sua porzione dei dati e produce un altro insieme intermedio come output;

2. **Shuffle**, fase intermedia in cui vengono distribuiti i dati in base alla chiave di input e aggregati insieme;

3. **Reduce**, vengono processati i dati in parallelo e restituisce il risultato.

Inizialmente viene diviso il file di input in blocchi di dimensione minore e ognuno di essi viene assegnato ad un **mapper** per essere processato. Quando tutti i mapper concludono i loro processi il framework ordina i risultati e li passa ai loro corrispettivi **reducer**. Quindi un reducer non può iniziare fino a quando un mapper sta ancora lavorando. Entrambi lavorano con coppie **chiave-valore** e ad un reduce vengono assegnati ed aggregati dati con la stessa chiave.
Possono essere applicate più fasi di Map e di Reduce, cioè accade che il risultato della fase di MapReduce venga passata con input ad una nuova fase MapReduce; fino a quando non si conclude.
Alla fine i risultati dell'ultimo reducer vengono riunite insieme in un unico output che rappresenta il risultato dell'intero processo.

2 Algoritmi

2.1 Algoritmo A-Priori

Algoritmo per trovare i frequent itemset che applica un doppio loop per ogni basket in cui nel primo va a generare le coppie e aggiunge uno al conteggio e poi per ogni volta che si trova la coppia nel basket si aumenta il conteggio di uno; in conclusione si va a controllare quali coppie hanno il conteggio maggiore o uguale alla *support threshold*.
In questo algorimto si fa un passo per ogni dimensione k dell'insieme. L'idea principale è la seguente: se non viene trovato un itemset frequente di dimensione k allora per la **monotonicità** sappiamo che non può esserci un frequent itemset più grande e quindi l'algoritmo si ferma.
Per passare dalla dimensione k all successiva dimensione $k+1$ dobbiamo sapere che per ogni dimension k ci sono due insiemi di itemset:

1. C_k che è l'insieme dei *candidate itemset* di dimensione k; gli itemset che dobbiamo contare per determinare se loro sono effettivamente frequenti;

2. L_k che è l'insieme dei frequent item di dimensione k che sono *veramente* frequenti.

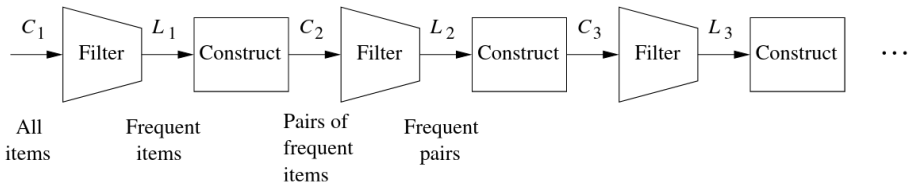

Figura 2.1: Passaggio tra C_k a L_k

Praticamente si parte da C_k e si applica il filtro per ottenere i frequent item che lo sono veramente per ottenere L_k e così via fino a quando non se ne trovano più.

2.1.1 Come funziona

Si parte da C_1 che contiene tutti gli itemset di dimensione 1 che non sono altro che gli item stessi. Il primo passo è quello di contare tutti gli item e quelli che hanno il conteggio almeno uguale alla threshold del supporto s compone l'insieme L_1 degli item frequenti.
L'insieme C_2 delle coppie candidate è l'insieme delle coppie dei quali entrambi gli elementi devono essere in L_1, cioè devono essere dei frequent item. Da notare che C_2 non viene costruito esplicitamente ma andiamo a costruirlo andando a testare la presenza degli elementi della varie coppie in L_1. Il secondo passo poi della A-Priori conta tutte le coppie candidate e determina quali appaiono almeno s volte. Queste formeranno il set L_2.
L'insieme C_3 delle triple candidate viene costruito implicitamente come l'insieme di triple in cui 2 elementi sono in L_2. Quindi diciamo che possiamo contare gli insiemi di dimensione k usando le tuple con $k+1$ componenti; dove l'ultima componente è il conteggio e le prime k componenti sono il codice degli item, ordinati.
Per trovare L_3 facciamo un terzo passaggio per il file che contiene i basket. Per ogni basket dobbiamo soltanto guardare quali item sono in L_1. Per questi item poi possiamo

esaminare ogni coppia e determinare se sono in L_2. Ogni elemento del basket che non appare in almeno due coppie frequenti, entrambi i quali consistono in item nel basket, non possono fare parte della tripla frequente che il basket contiene. In questo modo, abbiamo una ricerca abbastanza limitata di triple che sono sia contenute nel basket sia candidate in C3. Ad ogni tripla trovata viene aggiunto 1 al suo conteggio.
La costruzione della collezione dei frequent itemset procede fino a quando si vuole oppure finisce quando non si trovano nuovi itemset frequenti.

2.1.2 Riassumendo

1. Definire C_k che contiene tutti gli itemset di dimensione k, ogni $k-1$ dei quali è un itemset in L_{k-1};

2. Trovare L_k passando tra i basket e contando tutti e soli gli itemset di dimensione k che sono in C_k. Gli itemset che hanno un conteggio almeno s saranno in L_k.

2.2 Algoritmo S.O.N.

L'utilizzo di altri algoritmi per il problema degli itemset frequenti può portare ad ottenere *falsi positivi* e *falsi negativi* per esempio il **Randomize Algorithm** in cui si divide il file contenente i basket in diverse partizioni per eseguire la ricerca.
Un itemset è un **falso negativo** se risulta frequente sull'intero campione dei basket ma non lo è nel campione generato, mentre è un **falso positivo** se risulta positivo nel campione generato ma non nell'intero campione.
L'**algoritmo S.O.N.** va ad eliminare sia i falsi positivi che i falsi negativi.
L'idea consiste nel *dividere* il file di input in porzioni, detti **chunk**. Ogni chunk viene trattato come un campione e viene eseguito un algoritmo per trovare i frequent itemset per ogni chunk. In questo progetto ho scelto di utilizzare l'*algoritmo A-Priori* per questa parte perché mi è risultato più chiaro e l'ho avevo già affrontato nel corso di Estrazione e integrazione di conoscenza dei dati.
Viene utilizzata *ps* come *threshold* se ogni chunk è una frazione p dell'intero file e s è la *threshold* del supporto. Vengono poi salvati su disco tutti i frequent itemset trovati per ogni chunk.
Successivamente si uniscono tutti gli item frequenti trovati per uno o più chunk e questi saranno gli **itemset candidati**.

Un itemset che è frequente nell'intero dataset è frequente anche in almeno un chunk.

Per questo motivo non ci sono *falsi negativi*, mentre i *falsi positivi* si possono eliminare andando a contare nell'intero dataset tutti gli itemset che sono identificati come frequenti nel campione. Si andranno a tenere soltanto gli itemset frequenti del chunk che lo sono anche nell'intero dataset.

2.3 Algoritmo S.O.N. in MapReduce

Per la sua struttura l'algoritmo S.O.N. si presta alla perfezione per uno sviluppo **MapReduce**, in quanto ogni chunk può essere processato in parallelo e i frequent itemset trovati per ogni chunk possono essere combinati e formare gli itemset candidati. Poi

si possono distribuire gli insiemi candidati in più processori e ogni processore calcola il supporto per ogni candidato su un sottoinsieme dei basket e infine somma i supporti per ottenere il supporto dei candidati sull'intero dataset.
Questo viene eseguito con due passi di *MapReduce*:

1. trovare l'insieme degli **itemset candidati**;

2. trovare i **veri** itemset frequenti.

2.3.1 Prima Funzione di Map

Prende in input un sottoinsieme dei basket e trova gli itemset frequenti usando un algoritmo per trovarli. Solitamente si prende come sottoinsieme una frazione p dei basket e la threshold che si utilizza per il supporto dei sottoinsiemi è ps.
L'output è una coppia chiave-valore (F,1) dove F è un itemset frequente per il campione selezionato.

2.3.2 Prima Funzione di Reduce

Ad ogni Reduce Task viene assegnato un insieme di chiavi che sono itemset. Il task di reduce produce quelle chiavi (itemset) che compaiono una o più volte. Quindi l'output della prima fase del reduce sono gli **itemset candidati**.

2.3.3 Seconda Funzione di Map

I task del Map per la seconda fase prendono in input l'output del reduce della prima fase (gli itemset candidati) e una porzione del dataset (chunk). Ogni task conta il numero di occorrenze di ogni itemset candidato tra i basket della porzione del dataset che è stata passata come input.
L'output è un insieme di coppie chiave-valore (C, v) dove C è uno degli insiemi candidati e v è il supporto di esso tra i basket presenti nella porzione passata in input al task di map.

2.3.4 Seconda Funzione di Reduce

I task di Reduce prendono in input gli itemset che sono dati come chiave-valore dal Map e sommano i loro valori associati per ogni chiave. Il risultato è il **supporto totale** per ogni itemset che il task di Reduce sta elaborando. Gli itemset la cui somma è almeno s (la treshold del supporto dell'intero datset) sono frequenti nell'intero dataset. Quindi l'output di ogni task sono questi itemset e i loro conteggi.

2.3.5 Fase Finale

Alla fine abbiamo gli itemset che sono frequenti nell'intero dataset e cioè hanno un supporto che supera o è uguale al supporto minimo.

3 Scelte

In questa sezione vengono spiegate alcune scelte effettuate durante la progettazione e lo sviluppo dell'algoritmo. Altre decisioni vengono riportate durante l'esposizione del codice.

3.1 Spark

Le due opzioni per lo sviluppo dell'algoritmo SON con MapReduce erano o **Apache Hadoop** oppure **Spark**, ho scelto di utilizzare il secondo, in particolare lo sviluppo è stato effettuato con **pySpark**. Ho deciso questo framework perché l'ho trovato più intuitivo durante le lezioni di BigData e il linguaggio su cui si basa è *python*; inoltre Spark ha dei vantaggi rispetto ad Hadoop in quanto quest'ultimo è un po' limitato dal fatto che ogni job richiede tante letture e scritture su **HDFS**. Con Spark invece si legge soltando una volta da HDFS, cioè all'inizio e i dati vengono salvati nella memoria principale.
Il *problema* da gestire in Spark è la **volatilità** della memoria perché se accade quache problema si rischia di perdere tutto.
Praticamente in *Hadoop* si comincia elaborando il dato ma queste operazioni richiedono lettura e scrittura in HDFS; poi lancio il training di un modello e il risultato lo metto in HDFS e servirà per le diverse operazioni/query. Tutto questo richiede passaggi ad HDFS.
Invece *Spark* legge inizialmende da HDFS poi esegue il resto e infine scrive il risultato in HDFS, inoltre Spark lavora con **RDD**, **Resilient Distributed Datasets**, sono delle collezioni distribuite di dati che sono partizionate su diverse macchine e sanno in quali macchine si trovano; esse vengono generate da una lettura da HDFS oppure attraverso delle trasformazioni. Quindi tutte le elaborazioni che vengono fatte saranno basate su trasformazioni che si applicano al RDD.

3.2 A-Priori

Ho deciso di utilizzare l'**algoritmo A-Priori** per trovare gli itemset frequenti nella prima fase di Map dell'algoritmo SON perché esso mi è risultato più intuitivo rispetto ad altri e ho trovato più documentazioni e spiegazioni a riguardo ed è un algoritmo definito **unsupervised**.

4 Codice

In questa sezione viene rappresentato il codice per l'algoritmo A-Priori in python e le trasformazioni e azioni utilizzate in pySpark.
Il file contenente il codice è **spark_son_algorithm.ipynb**

4.1 Algoritmo A-Priori

Per l'implementazione dell'**algoritmo A-Priori** ho suddiviso il codice in 3 funzione:

1. `get_frequent_singletons`;

2. `getFrequents`;

3. `apriori_algorithm`

4.1.1 get_frequent_singletons

In questa funzione come prima cosa inizializzo un dict `candidates_dictionary` e una lista `singletons`, successivamente itero la lista che contiene i basket e per ogni item all'interno del basket controllo se è già stato inserito nel dizionario se non è presente allora lo inizializzo con valore 1 mentre se non è la prima volta che lo trova aggiunge 1 al valore che ha come chiave l'item.
Il dizionario costruito contiene una coppia chiave-valore in cui la chiave è l'item mentre il valore è il conteggio generato dal ciclo for. Una volta completato lo iteriamo e aggiungiamo alla lista creata inizialmente soltanto gli item che hanno il conteggio maggiore o uguale rispetto alla `local_threshold`, cioè al minimo supporto.
Quel che ritorna la funzione è una lista che contiene tutti gli **item frequenti**.

```
def get_frequent_singletons(baskets,local_threshold):
    """Funzione che calcola gli itemset di dimensione 1 che sono
    ↪ frequenti
    Input:
        - baskets, contiene gli itemset;
        - local_threshold, la threshold per filtrare gli item che
↪   hanno un supporto maggiore di essa.
    Output:
        - singletons, lista che contiene gli item frequenti per i
↪   basket passati in input.
    """
    candidates_dictionary = dict()
    singletons = list()

    for basket in baskets:
        for item in basket:
            if item not in candidates_dictionary.keys():
                candidates_dictionary[item] = 1
            else:
                candidates_dictionary[item] += 1

    for key,value in candidates_dictionary.items():
        if value >= local_threshold:
            singletons.append(key)

    return sorted(singletons)
```

Listing 1: getSingletons

4.1.2 getFrequents

Il seguente metodo calcola gli itemset frequenti per la dimensione che gli viene passata in input. Anche qui vengono inizializzate alcune variabili che tornano utili per lo sviluppo della funzione.
In pratica quello che viene svolto qui è il flusso della figura 2.1, cioè la creazione di C_k e L_k che sono corrispettivamente l'insieme degli itemset candidati come frequenti e l'insieme degli itemset veramente frequenti per la dimensione k; questo viene fatto per ogni dimensione fino a quando L_k non risulterà vuoto.
Premesse: `prev_freq` sono gli itemset frequenti trovati alla dimensione precedente, cioè L_{k-1}; `freq_itemsets` è la lista che viene popolata dagli itemset frequenti trovati nella dimensione attuale; `dict_table` è il dizionario che utilizzo per andare a salvare le coppie chiave-valore per memorizzare il conteggio di ogni itemset. Utilizzo inoltre la libreria `itertools` per creare le combinazioni.
Come prima cosa inizializzo due liste e un dizionario, poi si controlla la dimensione `dim` e se è uguale a 2 allora si aggiunge alla lista dei candidati, `candidates`, la combinazione di dimensione 2 degli itemset candidati di dimensione `dim` -1 cioè in questo caso gli itemset di dimensione 1. In questa casistica ricado alla prima esecuzione della funzione.
Se invece entriamo nell'`else` allora viene applicato il **Pruning**, cioè creo prima le tuple di dimensione `dim`, dalle combinazioni tra gli itemset frequenti nella dimensione precedente (L_{dim-1}), poi se la tupla appartiene già a `candidate`, la lista che rappresenta C_{dim}, cioè gli itemset frequenti nella dimensione `dim`, allora vado a quella successiva, altrimenti si procede generando le tuple di dimensione $dim - 1$ dalla combinazione della tupla candidata come frequente. Nel primo `if` all'interno del ciclo `for` filtro gli item andando a tenere soltanto quelli per cui l'intersezione tra le tuple al suo interno presentino elementi in comune, controllando che la lunghezza dell'insieme intersecato sia uguale a `dim - 2`.
A questo punto devo controllare se le ultime tuple generate, `pair`, appartengono a `prev_freq`, cioè a L_{dim-1} se lo sono allora posso aggiungere a C_{dim} la tupla `candidate_itemsets`.
Quando si completa questa procedura per tutte le tuple generate dalla prima combinazione appena entro nell'`else` si ha la lista `candidates`; viene applicato un ciclo su essa e per ogni itemset candidato (`candidate`) si calcola il numero di volte che appare nei basket, cioè il suo *supporto*.
Infine vado ad aggiungere a `freq_itemsets` soltanto gli itemset che hanno il supporto maggiore di `local_thershold`, cioè il minimo supporto. Il risultato di questa funzione è L_{dim}, gli itemset veramente frequenti alla dimension `dim`.

```
def getFrequents(baskets, prev_freq, dim,local_threshold):
    """Funzione la quale calcola gli itemset frequenti di dimensione
    ↪ maggiore di 1.
    Input:
        - basket, insieme di itemset su cui stiamo lavorando;
        - prev_freq, gli itemset frequenti di dimensione dim-1;
        - dim, la dimensione attuale;
        - local_threshold, minimo supporto che gli itemset devono
↪   superare o raggiungere per essere
            considerati frequenti
    Output:
        - freq_itemsets, lista degli itemset frequenti
    """
    candidates = list()
    dict_table = dict()
    freq_itemsets = list()
    if dim==2:
        table_C2 = itertools.combinations(prev_freq,dim)
        for element in table_C2:
            candidates.append(element)
    else:
        for item in itertools.combinations(prev_freq,2):
            if(len(set(item[0]).intersection(set(item[1]))) == dim-2):
                candidate_itemsets =
                ↪ tuple(set(item[0]).union(set(item[1])))
                if candidate_itemsets in candidates:
                    continue
                else:
                    pair = itertools.combinations(candidate_itemsets,d
                    ↪ im-1)

                    if set(pair).issubset(prev_freq):
                        candidates.append(candidate_itemsets)

    for candidate in candidates:
        for basket in baskets:
            if set(candidate).issubset(set(basket)):
                dict_table.setdefault(candidate,0)
                dict_table[candidate] +=1

    for key,value in dict_table.items():
        if value >= local_threshold:
            freq_itemsets.append(key)

    return sorted(freq_itemsets)
```

Listing 2: getFrequents

4.1.3 apriori_algorithm

La seguente è la funzione principale che richiama le altre, infatti qui nella prima parte andiamo a calcolare il minimo supporto e a calcolare gli item frequenti. Poi nella seconda parte c'è un ciclo `While` in cui all'interno calcolo gli itemset frequenti per ogni dimensione e lo faccio fino a quando non trovo più itemset frequenti.
Ritorna gli itemset che noi consideriamo **candidati**.
Questa funzione viene eseguita per ogni chunk che viene generato.
La variabile `gloabl_supp_threshold` è una variabile che dichiaro precedentemente.

```
def apriori_algorithm(chunk):
    """Funzione che applica l'algoritmo APriori.
    Input:
        - chunk: partizione del file di input che contiene i basket
    Output:
        - freq_itemsets, lista degli itemset candidati dopo
↪   l'applicazione dell'algoritmo A-Priori
    """
    baskets = list()
    freq_itemsets = list()
    current_freq = list()
    for c in chunk:
        baskets.append(c)
    dimension = 1
    local_supp_threshold = len(baskets) * global_supp_threshold
    freq_singletons =
    ↪  get_frequent_singletons(baskets,local_supp_threshold)

    for single in freq_singletons:
        freq_itemsets.append(single)

    dimension +=1

    current_freq = freq_singletons

    while True:
        #print(f'curr freq: {current_freq}')
        prev_frequents = current_freq
        #print(f'prev_frequents: {prev_frequents}')
        current_freq = getFrequents(baskets,prev_frequents,dimension,l
        ↪  ocal_supp_threshold)
        #print(f'curr freq dopo: {current_freq}')
        for item in current_freq:
            freq_itemsets.append(item)

        if len(current_freq) == 0:
            break
        dimension +=1

    return freq_itemsets
```

Listing 3: A-Priori Algorithm

4.2 Algoritmo SON

Di seguito andremo a mostrare l'implementazione **MapReduce** dell'algorimto S.O.N. e come anticipato nella presentazione dell'algoritmo, esso è strutturato in due fasi di MapReduce.

4.2.1 Prima Fase MapReduce

Con il seguente codice andiamo ad implementare la prima fase di MapReduce in cui otteniamo gli itemset candidati. Viene applicato l'algoritmo A Priori per calcolare gli itemset frequenti dai quali poi otteniamo gli itemset candidati.

```
candidatesItemsets = file.mapPartitions(apriori_algorithm).map(lambda
↪  item: (item,1))
```

Listing 4: Prima applicazione del Mapper

Successivamente passiamo da una RDD ad una lista applicando un'*azione*. La lista appena creata contiene come elementi delle coppie chiave-valore in cui il valore è 1.

```
candidatesItemsets_list = candidatesItemsets.collect()
```

Listing 5: Output Prima Fase di MapReduce

4.2.2 Seconda Fase di MapReduce

La stesura del codice continua con la seconda fase di MapReduce definendo una nuova funzione che conta le occorrenze di ogni itemset candidati all'interno del chunk passato come parametro.

```
def countOccurences(chunk,candidates):
    '''Funzione che conta le occorrenze degli itemset all'interno del
    ↪  chunk
        Input:
            - chunk, porzione del file di input;
            - candidates, lista che contine gli itemset.
        Output:
            - lista che contiene coppie chiave-valore dove la chiave
↪   è l'itemset e il valore è il suo conteggio
    '''
    itemsetOccurences = list()
    dictionary = dict()
    chunk_list = list(chunk)
    for basket in chunk_list:
        for candidate in candidates:
            if(set(candidate[0]).issubset(set(basket))):
                dictionary.setdefault(candidate[0],0)
                dictionary[candidate[0]] +=1

    for key,value in dictionary.items():
        itemsetOccurences.append((key,value))
    return itemsetOccurences
```

Listing 6: countOccurences

Si procede con l'applicazione del Map:

```
output_SecMapReduce = file.mapPartitions(lambda chunk:
↪   countOccurences(chunk,candidatesItemsets_list)).reduceByKey(lambda
↪   a,b: a+b)
```

Listing 7: Output Seconda Fase di MapReduce

Qui calcolo il numero di itemset candidati e il numero totale di basket:

```
num_itemsetCandidates = output_SecMapReduce.count()
num_tot_baskets = file.count()
```

Listing 8: Seconda parte Reduce

4.2.3 Parte Finale

Da qui in poi ci sono due strade, il calcolo degli itemset frequenti utilizzando la normalizzazione e non utilizzandola.

Normalizzazione:

```
def normalize_values(item,min_value,max_value):
    '''Funzione che normalizza i valori:
        Input:
            - item, l'item da normalizzare che contiene chiave e
↪   valore;
            - min_value, il valore minimo negli itemset candidati;
            - max_value, il valore minimo negli itemset candidati.
        Output:
            - ritorna una tupla che contiene chiave e valore
↪   normalizzato
    '''
    norm_value = (item[1]-min_value)/(max_value-min_value)
    return (item[0],norm_value)
```

Listing 9: Funzione di normalizzazione

Decido si settare come supporto minimo per il caso normalizzato come la `global_supp_threshold` settata ad inizio sviluppo:

```
minSupportNormalized = global_supp_threshold
```

Listing 10: Settaggio minimo supporto

Calcoliamo alla fine gli itemset frequenti andando ad applicare un filtro per mantenere come frequenti gli itemset che hanno il supporto maggiore o uguale al supporto minimo. Inoltre si salva il risultato su una variabile `frequentItemset_normalizzato` con il passaggio da RDD a lista.

```
outputFinale_normalizzato = normalized_secReduce.filter(lambda item:
↪   item[1] >= minSupportNormalized)\
    .map(lambda item:item[0]).sortBy(lambda item: len(item))
frequentItemset_normalizzato = outputFinale_normalizzato.collect()
```

Listing 11: Settaggio minimo supporto

Senza Normalizzare:
Calcolo il supporto minimo prendendo il valore **medio** tra tutti i supporti (occorrenze) degli itemset candidati. Concludiamo con il passaggio da RDD a lista che contiene gli itemset frequenti calcolati dall'algoritmo SON.

```
def getminSuppwith_avg(candidates,num_candidates):
    '''Funzione che ritorna il supporto minimo andando
        a calcolare il valore medio.
        Input:
            - candidates, itemset candidati;
            - num_candidates;
        Outpur:
            supporto minimo'''
    sums = 0
    for c in candidates.collect():
        sums+= c[1]

    return int(sums/num_candidates)

minSupport =
 ↪  getminSuppwith_avg(output_SecMapReduce,num_itemsetCandidates)
```

Listing 12: Funzione per il calcolo del supporto minimo e utilizzo

Applichiamo lo stesso filtro descritto sopra ma con un diverso supporto minimo.

```
outputFinale = output_SecMapReduce.filter(lambda item: item[1] >=
 ↪  minSupport)\
    .map(lambda item:item[0]).sortBy(lambda item: len(item))

frequentItemsets = outputFinale.collect()
```

Listing 13: Frequent Itemsets

Infine nell'ultima parte del codice vado a salvare il risultato su un file per entrambi i casi.

```
name = name_file.split('.')[0]
f = open('./output/output_frequentItemsets_' + name +
↪  str(num_partition) +'.txt', 'w')
for item in frequentItemsets:
        if type(item) is tuple:
            length=len(item)
            f.write("(")
            for i in range(0,length-1):
                f.write(str(item[i])+",")
            f.write(str(item[length-1])+")\n")
        else:
            f.write(item + "\n")
f.close()
```

Listing 14: Salvataggio File

5 Risultati

In questo capitolo vengono presentati i risultati ottenuto dallo sviluppo dell'algoritmo SON attraverso l'utilizzo di pySpark.
Verranno confrontati i risultati dello sviluppo in *parallelo*, cioè quello portato avanti nell'ambiente Spark, e quello ottenuto dallo sviluppo in locale. Per quest'ultimo ho riscritto il codice ma senza l'utilizzo di Spark e il file contenete il codice è **spark_son_algorithm.ipynb**.
I processi sono stati lanciati su una macchina con processore *AMD Ryzen 9 3900X 12-Core Processor* con 24 core.

5.1 Normalizzazione e non

Come prima cosa si confrontano i risultati dello sviluppo parallelo tra i due casi sviluppati, cioè con normalizzazione e senza normalizzazione.
Il dataset utilizzato è `accidents.dat`, il numero di partizioni è 12 e come threshold per il caso della normalizzazione è settato a 0.4, mentre per il caso normale il minimo supporto è 942365. Il numero di itemset frequenti è ovviamente differente perché si utilizzano due threshold diverse, nel caso della normalizzazione cisono 682 itemset frequenti mentre nell'altro ce ne sono 1900.

5.2 Confronto tra sviluppo Parallelo e Locale

Il confronto viene fatto senza Normalizzare i valori.
Dati **primo caso**:

1. numero di partizioni, 12;

2. dataset, accidents.dat;

Il tempo necessario per l'esecuzione dell'algoritmo a priori per i vari chunk nello sviluppo parallelo ha richiesto **2 minuti** mentre per lo sviluppo locale ne ha richiesti **19** e per

l'applicazione dell'algoritmo SON lo sviluppo locale ad un certo punto si è bloccato.

Dati **secondo caso**:

1. numero di partizioni, 4;
2. dataset, accidents.dat;
3. global_support_threshold è 0.4

Numero di itemset frequenti è 763 ci ha messo un totale di 5 minuti per l'applicazione dell'algoritmo SON.

Dati **terzo caso**:

1. numero di partizioni, 24;
2. dataset, accidents.dat;
3. global_support_threshold è 0.4

Numero di itemset frequenti è 995 ci ha messo un totale di 7 minuti per l'applicazione dell'algoritmo SON.

6 Conclusioni

Il codice implementato riesce a trovare i frequent itemset partendo da un dataset. In base al minimo supporto scelto il risultato ovviamente cambia. Ho riscontrato problemi con file troppo grandi anche utilizzando 24 partitioner.
I risultati vengono salvati su dei file txt e possono essere visualizzati nella cartella output.

6.1 Difficoltà

Le maggiori difficoltà le ho incontrate all'inizio del progetto dove sono andato a studiare e capire per prima cosa cosa si intende per frequent itemset e poi il funzionamento dell'algoritmo A-Priori e dell'algoritmo SON.
Poi mi sono bloccato durante la scrittura del codice perché non riuscivo a capire bene come suddividere nel codice le diverse fasi di Map e di Reduce.
Un altro punto in cui ho avuto qualche dubbio qui ma anche in passato lavorando su questi argomenti è sulla scelta del supporto da utilizzare come soglia.