INF 553 – Fall 2017
Assignment 2: Frequent Itemsets

Deadline: 10/11 2017 11:59 PM PST

Assignment Overview

This assignment contains one main algorithm. You will implement the SON algorithm using the Apache Spark Framework. **Spark 1.6 should be used for this HW. For Scala implementation please use Scala version 2.10 and sbt version 0.13.6 to build jar files**. You will use different support threshold with a varied range. This will help you to test, develop and optimize your algorithm given the number of records at hand. More details on the structure of the dataset and instructions on how to use the input files will be explained in detail in the following sections. The goal of this assignment is to make you understand how you can apply the frequent itemset algorithms you have learned in class on a large number of data and more importantly how you can make your implementation more performant and efficient in a distributed environment.

Write your own code!

For this assignment to be an effective learning experience, you must write your own code!
Do not share code with other students in the class!!
Here's why:

- The most obvious reason is that it will be a huge temptation to cheat: if you include code written by anyone else in your solution to the assignment, you will be cheating. As mentioned in the syllabus, this is a very serious offense, and may lead to you failing the class.

- However, even if you do not directly include any code you look at in your solution, it surely will influence your coding. Put another way, it will short-circuit the process of you figuring out how to solve the problem, and will thus decrease how much you learn.

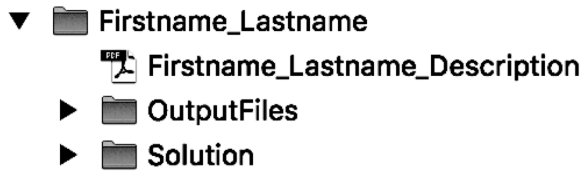
So, just don't look on the web for any code relevant to these problems. Don't do it.

Submission Details

For this assignment you will need to turn in a Python or Scala program depending on your language of preference. We will test your code using the same dataset but with different support thresholds values. **This assignment will surely need some time to be implemented so please plan accordingly and start early!**

Your submission must be a .zip file with name: **<Firstname>_<Lastname>_hw2.zip**. The structure of your submission should be identical as shown below. The Firstname_Lastname_Description.pdf file contains helpful instructions on how to run your code along with other necessary information as described

in the following sections. The *OutputFiles* directory contains the deliverable output files for each problem and the *Solution* directory contains your source code.



SON Algorithm

In this assignment, we implement the SON Algorithm to solve every case on top of Apache Spark Framework. We will rely on the fact that SON can process chunks of data in order to identify the frequent itemsets. You will need to find **all the possible combinations of the frequent itemsets** for any given input file that follows the format of the MovieLens Ratings Dataset ml-1m. In order to accomplish this task, you need to read Chapter 6 from the Mining of Massive Datasets book and concentrate on section 6.4 – Limited-Pass Algorithms. Inside the Firstname_Lastname_Description.pdf file we need you to describe the approach you used for your program. Specifically, in order to process each chunk, you **should use A-Priori** for counting frequent item-set in each phase.

At the end of the assignment, Appendix A provides some more guidelines that will help you with the implementation and Appendix B specifies how to organize your Description pdf file.

For assignment 1 you used the Spark framework and most probably at this point you have a better understanding of the MapReduce operations. You can write your program in Python or Scala. For this assignment, you will need to find the collection of frequent itemsets of rated movies using the MovieLens dataset with which you are already familiar from homework 1. You will need to compute the frequent itemsets using SON algorithm, for the same dataset as Assignment 1 that is **ml-1m (**ml-1m.zip)

You can also download the dataset. Once you extract the zip archives you will find multiple data files. From those files for this assignment we only need the *ratings.dat and users.dat* file from zip archive.

Case 1

We would like to calculate the combinations of frequent movies (as singletons, pairs, triples, etc…) that were rated by **male users** and are qualified as frequent given **a support threshold value**.

In order to apply this computation, we will need to create a basket for each male user containing the ids of the movies that were rated by this male user. If a movie was rated more than one time from a user, we consider that this movie was rated only once. More specifically, the movie ids are unique within each basket. The generated baskets are similar to:

Male-user1 = (movie$_{11}$, movie$_{12}$, movie$_{13}$, …)
Male-user2 = (movie$_{21}$, movie$_{22}$, movie$_{23}$, …)
Male-user3 = (movie$_{31}$, movie$_{32}$, movie$_{33}$, …)
…

Case 2

In the second case we want to calculate the combinations of **frequent female users** (as singletons, pairs, triples, etc…) who rated the movies. The frequent combination of female users has to be calculated on the basis of the support **threshold value**.

In order to apply this computation, we will need to create a basket for each movie that has been rated by the female users. More specifically, the female users ids are unique within each basket. The generated baskets are similar to:

Movie 1 = (female-user$_1$, female-user$_2$, female-user$_3$, …)
Movie 2 = (female-user$_3$, female-user$_4$, female-user$_5$, …)
Movie 3 = (female-user$_1$, female-user$_2$, female-user$_5$, …)
…

Finally, in the following section, we will describe explicitly how you should run your program, and what should be the format of your expected output. Everything that is described in section must be applied to both the cases

Implementing SON using Spark with ml-1m dataset

Under the */Data* folder of the assignment you will the actual dataset. Following is the description for the execution requirements.

Execution Requirements

Input Arguments:

1. **Case Number:** An integer value specifying which case from the ones we just described we want to compute the frequent itemsets. The input is a integer value **1 for case 1** and **2 for case 2**.

2. **Users.dat:** This is the path to the input users file containing all the user's information. Each line corresponds to a user. Each line has items that are (: :) separated.

3. **Ratings.dat:** This is the path to the input ratings file containing all the ratings information. Each line corresponds to a rating given by the user for the corresponding movie. Each line has items that are (: :) separated.

4. **Support:** Integer that defines the minimum count to qualify as a frequent itemset.

Output:

A file in the format shown in the snapshot of the Execution Example section below. In particular, for each line you should output the frequent itemsets you found for the current combination followed by an empty line after each combination. The printed itemsets must be sorted in ascending order. A high-level description of this format is:

(frequent_singleton1), (frequent_singleton2), …, (frequent_singletonK)

(frequent_pair1), (frequent_pair2), …, (frequent_pairM)

(frequent_triple1), (frequent_triple2), …, (frequent_tripleN)
…

Execution Example

The first argument passed to our program in the below execution is the case number we want to run the algorithm against. The second input is the path to the ratings input file and the third is the support threshold value. Following we present examples of how you can run your program with spark submit both when your application is a Scala program or a Python script.

A. **Example of running a Scala application with spark-submit:**
Notice that the argument class of the spark-submit specifies the main class of your application and it is followed by the jar file of the application.

For Case 1
Priyambadas-MacBook-Pro: spark-2.1 priyambadajain$ ./bin/spark-submit --class Priyambada_Jain_SON --master local[*] Priyambada_Jain_SON.jar 1 ratings.dat users.dat 1200

For Case 2

Priyambadas-MacBook-Pro: spark-2.1 priyambadajain$ ./bin/spark-submit --class Priyambada_Jain_SON --master local[*] Priyambada_Jain_SON.jar 2 ratings.dat users.dat 500

B. **Example of running a Python application with spark-submit:**

Case 1

Priyambadas-MacBook-Pro: spark-2.1 priyambadajain$ ./bin/spark-submit Priyambada_Jain_SON.py 1 ratings.dat users.dat 1200

Case 2

Priyambadas-MacBook-Pro: spark-2.1 priyambadajain$ ./bin/spark-submit Priyambada_Jain_SON.py 2 ratings.dat users.dat 500

The solution of the above execution for case 1 is similar to the following snapshot.

Solution of Case 1, with input case 1 , input files ratings.dat and users.dat and support threshold equal to 1300:

```
(1), (50), (110), (260), (296), (318), (356), (457), (480), (527), (541), (589), (593),
(608), (780), (858), (924), (1036), (1097), (1127), (1196), (1197), (1198), (1200), (1210),
(1214), (1221), (1240), (1259), (1265), (1270), (1387), (1580), (1610), (1617), (2000),
(2028), (2396), (2571), (2628), (2716), (2762), (2791), (2858), (2916), (2987), (2997),
(3175), (3578)
(110, 480), (110, 589), (110, 1196), (110, 2028), (260, 480), (260, 589), (260, 858), (260,
1097), (260, 1196), (260, 1198), (260, 1210), (260, 1214), (260, 1240), (260, 1270), (260,
1580), (260, 2028), (260, 2571), (260, 2628), (260, 2858), (260, 2916), (480, 589), (480,
1196), (480, 1210), (480, 1580), (480, 2571), (480, 2858), (480, 2916), (589, 1196), (589,
1198), (589, 1210), (589, 1240), (589, 1270), (589, 1580), (589, 2028), (589, 2571), (589,
2628), (589, 2858), (589, 2916), (593, 608), (593, 2858), (608, 2858), (1097, 1196), (1196,
1197), (1196, 1198), (1196, 1200), (1196, 1210), (1196, 1214), (1196, 1240), (1196, 1270),
(1196, 1580), (1196, 2028), (1196, 2571), (1196, 2628), (1196, 2858), (1196, 2916), (1198,
1210), (1210, 1240), (1210, 1270), (1210, 1580), (1210, 2028), (1210, 2571), (1210, 2858),
(1240, 2571), (1580, 2571), (1580, 2628), (1580, 2916), (2028, 2571), (2028, 2858), (2571,
2628), (2571, 2858), (2571, 2916), (2762, 2858), (2858, 2997)
(260, 480, 1196), (260, 589, 1196), (260, 589, 1210), (260, 589, 2571), (260, 1196, 1198),
(260, 1196, 1210), (260, 1196, 1240), (260, 1196, 1270), (260, 1196, 2571), (260, 1210,
2571), (480, 589, 1580), (480, 589, 2571), (589, 1196, 1210), (589, 1196, 2571), (589, 1580,
2571), (1196, 1198, 1210), (1196, 1210, 2571)
```

Deliverables

1. **Script or Jar File and Source Code:**
   Please name your Python script as: <firstname>_<lastname>_SON.py.
   Or if you submit a jar file as: <firstname>_<lastname>_SON.jar.

   The python script or the .jar file of your implementation should be inside the *Solution* directory of your submission. You must also include a directory, any directory name is fine, with your source code inside *Solutions*.

2. **Output Files:**
   We need two output files for each case.
   For case 1, run your program against *ml-1m* dataset with s**upport 1200** and support **1300**.
   For case 2, run your program against *ml-1m* dataset with **support 500** and **support 600**.

   The format of the output should be exactly the same as the above snapshot for both cases.

   The names of the output files should be as:

   <firstname>_<lastname>_SON.case1_1200.txt
   <firstname>_<lastname>_SON.case1_1300.txt
   <firstname>_<lastname>_SON.case2_500.txt
   <firstname>_<lastname>_SON.case2_600.txt

   The above output files should be placed inside the *OutputFiles* directory of your submission.

3. **Description:**
   Inside the Firstname_LastName_Description pdf document please write the command

line that you used with spark-submit in order to run your code. If it is a jar file, please specify the name of the main class of your app as shown in the above snapshots. We will use this in order to rerun your code against different support values if needed.

4. **Time Complexity:**
Your program should follow time limit for each case and support threshold as described.

CASE 1		**CASE 2**	
Support Threshold	**Execution Time**	**Support Threshold**	**Execution Time**
1200	<120secs	500	<60secs
1300	<65 secs	600	<50 secs

General Instructions:

1. Make sure your code compiles before submitting
2. Make sure to follow the output format and the naming format.
3. For Scala implementation please restrict to include jars and class files (whole project files are not needed).

Grading Criteria:

1. If your programs cannot be run with the commands you provide, your submission will be graded based on the result files you submit and 20% penalty for it.
2. If the files generated by your programs are not sorted based on the specifications, there will be 20% penalty.
3. If your program generates more than one file, there will be 20% penalty.
4. **If you don't provide the source code and just the .jar file in case of a Scala application there will be 60% penalty.**
5. **If your submission does not state inside the Description pdf file how to run your code, which approach you followed to implement your algorithm there will be a penalty of 30%.**
6. There will be 20% penalty for late submission
7. Also, as described for Scala implementation **10% bonus will be awarded.**

APPENDIX A

- You need to take into account the Monotonicity of the Itemsets
- You need to leverage Spark capabilities of processing partitions/chunks of data and analyze the data within each partition.
- You need to reduce the support threshold according to the size of your partitions.
- You should emit appropriate (key, value) pairs in order to speed up the computation time.
- Try to avoid data shuffling during your execution.

Pay great attention on the thresholds number for each case. The lower the threshold the more the computation. Do not try arbitrary threshold values. Try testing values within the given ranges.

APPENDIX B

Please include the following information inside your description document.

- Succinctly describe your approach to implement the algorithm.
- Command line command to execute your program