Brand, Alena Nicole

CS 4710 introduction to data mining

Personal Implementation: Apriori Algorithm

comparison of the effectiveness of varying implementations of the apriori algorithm

Contents

[1. Algorithm 2](#_Toc6920914)

[1.1. Overview 2](#_Toc6920915)

[2. Implementation 2](#_Toc6920916)

[2.1. General Notes 2](#_Toc6920917)

[2.2. Creating the Database 2](#_Toc6920918)

[2.3. Finding Candidate One Itemsets 2](#_Toc6920919)

[2.4. Finding Frequent Itemsets 2](#_Toc6920920)

[2.5. Finding Candidate Itemsets 3](#_Toc6920921)

[2.5.1. Finding Candidate Two Itemsets 3](#_Toc6920922)

[2.5.2. Finding k Candidate Itemsets 4](#_Toc6920923)

[2.6. Counting Candidate Itemsets 4](#_Toc6920924)

[2.7. Pruning Lk Itemsets 4](#_Toc6920925)

[2.8. The Driver Program 4](#_Toc6920926)

[2.8.1. Timing the Algorithm 5](#_Toc6920927)

[2.9. Printing Lists to the Screen 5](#_Toc6920928)

[3. Results 5](#_Toc6920929)

[4. Discussion 8](#_Toc6920930)

[4.1. Reasons for using Python 8](#_Toc6920931)

[4.2. Comparison with 1995 Implementation 8](#_Toc6920932)

[4.3. Comparison with Python Implementations 9](#_Toc6920933)

[4.3.1. Modifications and Notes 9](#_Toc6920934)

[4.3.2. Comparison with the Python 2 Hash Tree Implementation 9](#_Toc6920935)

[4.3.3. Comparison with the Python 2 Non-Hash Tree Implementation 9](#_Toc6920936)

[4.4. Overall Effectiveness of Algorithm 10](#_Toc6920937)

1. Algorithm
   1. Overview

In this paper, a variation on the Apriori algorithm is presented. The overall idea for the algorithm was taken from “Fast Algorithms for Mining Association Rules” by Rakesh Agrawal and Ramakrishnan Srikant. The Apriori algorithm is described by the following steps or procedures:

1. Find frequent one itemsets in the first pass and put them in L1
2. Pairing frequent one itemsets to find candidate two itemsets, during the second pass
3. Find frequent two itemsets and putting them in L2, at the very end of the second pass
4. Perform the joining step for every kth pass to determine candidate k itemsets, for k > 2 passes
5. Find frequent k itemsets to be inserted into Lk
6. Prune itemsets from Lk to finish the kth pass
7. Repeat steps four through six, until the length of Lk < 2
8. Implementation
   1. General Notes

Version 3 of Python was chosen to implement this specific instance of the Apriori algorithm. The reasons for this will be discussed in the implementation and the discussion sections. For ease of reading, variable names and function names were attempted to be named after important concepts in the Apriori algorithm. This would include things like L1, L2, Lk, c1, c2, ck, k, and itemset. These naming conventions also made programming much easier because the goals for each specific function were well understood based on the Apriori algorithm.

* 1. Creating the Database

The first topic to be discussed is how the database is read into my program for later use by the Apriori algorithm. This step is very important, although it seems trivial. Firstly, the data in the file was separated by new line characters, so the transactions could be separated. Then each transaction was trimmed to remove the extra whitespace at the end of each transmission in the file. Then each transaction was split by spaces, or tabs to differentiate the items in a transaction. The whole database is stored in a list. This list contains sublists that represent each transaction. Please reference the create\_transaction\_database function in Apriori.py.

This was the easiest storage method that could be determined since the items were added to the database list as the file was read. It should also be noted that list comprehension was used to create the database. We will now relate the steps of the Apriori algorithm to this specific implementation.

* 1. Finding Candidate One Itemsets

To find candidate one itemsets, the database was scanned. Every time an item was scanned it was checked against the dictionary to see if it was already found. Every time a new item was found, it was added to a dictionary and its count (dictionary value) was set to one. If an item was already in the dictionary, it's count was incremented. This is very efficient for the first pass, since one does not have to union (k = 2) or join (k > 2) the itemsets, since the algorithm is only trying to obtain possible one itemsets. Please reference the find\_one\_itemsets function in Apriori.py.

* 1. Finding Frequent Itemsets

Determining frequent itemsets is the same for any value of k, so it will only be discussed once. The candidate itemset counts are compared to the minimum support percentage. If the percent derived count for an itemset is greater than or equal to the minimum support, the itemset and its count, in the form of nested tuples, are added to a list. This first item in the list is a tuple representing the itemset. The second item in the tuple is its respective count.

The only difference between finding frequent one itemsets and frequent k itemsets is that after the frequent k itemsets are found, they are sorted by the numbers (items) in their set. This is so the joining step works appropriately when trying to find future candidate itemsets. Please reference the find\_frequent\_itemsets function in Apriori.py.

* 1. Finding Candidate Itemsets

Generally speaking, this implementation of the Apriori algorithm is very similar to the algorithm that was first implemented in the paper mentioned above. However, there is one vital difference that may lead to some performance issues. It was decided that this algorithm should use a hash table, instead of a hash tree. The main reason behind this decision is the lack of resources that are available for one to easily implement a hash tree. Most of the information found online was related to Merkel trees. Merkel trees are a more specialized version of a generic hash tree. They are used in the field of cryptography, so this was not useful for this implementation.

The search for a general hash tree only yielded one option that could be used in the Apriori algorithm It can be found here: https://raw.githubusercontent.com/omjego/AR-Mining-Hash-Tree/master/apriori\_with\_htree.py. However as one can see, this specific implementation of a hash tree is very interconnected with the author's implementation of the Apriori algorithm. Because of this, this code was decidedly not useful because it would be difficult to provide a new implementation of the Apriori algorithm around it.

The second reason as to why a hash tree was deemed not necessary is because of how Python implements it's dictionary data type. A dictionary data type is a hash table that stores keys and values. It was decided that this could be used to store candidate itemset counts in a combination that is both simple and efficient. Since the dictionary is a built-in data type in Python, the act of maintaining it to store itemset counts is very simple. Also, since dictionaries employ hashing, they are still substantially more efficient than using other data types to store candidate itemsets, like 2D arrays.

In this implementation, the key of the dictionary is a tuple representing an itemset. The itemsets had to be stored as tuples because dictionary keys must be immutable. The value of each entry is the respective count of each candidate itemset. Initially, the values of all candidates are set to 0, so they can be easily incremented when the database is later scanned to obtain itemset counts. It should be noted that finding candidate two itemsets and candidate k itemsets are two separate functions because getting candidate to itemsets can be done by using the union, whereas k itemsets requires join.

* + 1. Finding Candidate Two Itemsets

For candidate to itemsets, the only thing that needs to be done is finding all possible two number combinations of every item in L1. This was done using Python’s ittertools library and the combinations function. The combinations function takes two parameters; a list or set of items that will be combined, in the amount of items in a combination set. Thankfully, the combinations function returns a list of tuples, so that tuples could be added directly into the dictionary through simple iteration. Please reference the find\_two\_itemsets function in Apriori.py.

* + 1. Finding k Candidate Itemsets

The function for finding candidate k itemsets is slightly more difficult. In order to accomplish this two indexes must iterate through the previous Lk list, one after the other. Then it is checked to see if the first k -2 items of those two itemsets in Lk are equivalent. If they are a tuple is inserted into a candidate itemset dictionary which contains all of the items in the itemset at the first index along with the last item in the itemset at the second index. The count of this new itemset is set to zero.

For example, if one was looking for candidate L3 itemsets and L2 contained {1, 2}, {1, 3}, and {2, 3}, the first index would be set to zero and the second index would be set to one. Therefore, the first index would be pointing to {1, 2} in the second index would be pointing to {1, 3}. Since both of these itemsets have a first value of 1, 1 and 2 are added to the candidate three itemset along with the last item two pointed by the second index which is 3. Therefore, the candidate 3 itemset is {1, 2, 3}. After this item is added to c3, the second index pointer moves until it reaches the end of the list. At this time the first index will be incremented in the second index will start again one ahead of the first index. This means that all the itemsets will be visited. Please reference the find\_k\_itemsets function in Apriori.py.

* 1. Counting Candidate Itemsets

In order to count the candidate itemsets, which will later be used to determine if the itemsets are frequent is again accomplished by using the ittertools combinations function. In order to determine if an itemset is contained within the transaction, every transaction in the database is looped through to determine all possible combinations of size k. Once the combinations for specific transaction are found, every item in ck is checked to see if it is a subset of the possible combinations of that transaction by using the issubset function provided by Python. If it is a subset of a transaction, that itemset is found in the ck dictionary and its value is incremented. Please reference the count\_itemsets function in Apriori.py.

* 1. Pruning Lk Itemsets

For the pruning itemsets function, every necessary combination of Lk itemsets is determined by using the ittertools combination function on every itemset in Lk. In order to determine if all the necessary subsets of found, they’re check to see if they are in the Lk - 1 itemset by using the issubset function that Python provides. Accounting is used to determine how many subsets were found in the previous list. If this number is equivalent to the number of necessary subsets, the item is kept in Lk. If the correct number of necessary subsets were not removed, that means not all the subsets were present, so the itemset is removed from Lk. Please reference the prune\_list function in Apriori.py.

* 1. The Driver Program

The driver program, which can be found by looking at Driver.py is what allows the Apriori algorithm to execute. The driver program cycles through these main steps:

1. Read in the database
2. Find and count candidate one itemsets
3. Determine frequent one itemsets
4. Determine candidate two itemsets
5. Count the occurrences of candidate two itemsets
6. Determine frequent two itemsets
7. Read in the database
8. Find and count candidate one itemsets
9. Determine frequent one itemsets
10. Determine candidate two itemsets
11. Count the occurrences of candidate two itemsets
12. Determine frequent two itemsets

The driver program was implemented in this way to account for the differences in methodologies when determining L1, L2, and Lk. L3 was separated out to account for doing the pruning step with L2 instead of the previous list parameter. Also, when using the pruning function, the previous list had to be a deep copy of Lk, so the values of both lists were not modified together. Python by default uses shallow copying, so the copy library was implemented, and the deep copy function was used to copy Lk into the previous list variable.

1. If the size of L2 is greater than one:
   1. Determine candidate three itemsets
   2. Count the occurrences of candidate three itemsets
   3. Determine frequent three itemsets
   4. Prune L3 by using L2
   5. Set the value of L3 to a variable called prev\_list
2. if the size of Lk is greater than one:
   1. Determine candidate k itemsets (for k > 4)
   2. Count the occurrences of candidate k itemsets
   3. Determine frequent k itemsets
   4. Prune Lk by using prev\_list
   5. Set the value of Lk to a variable called prev\_list
   6. Increment k
      1. Timing the Algorithm

In order to time the algorithm, which will eventually be used to determine its effectiveness, the time library from Python was imported. At the start of each run, the start time was taken using time.time(). When the algorithm finished running, the total time taken by the algorithm was calculated by subtracting time.time() from the start time. This information, along with the minimum support value was output to a text file.

For this algorithm test the minimum support started at a value of two. At the end of each run after the output was written, the minimum support was decremented by .5, or .25 depending on its current value. The algorithm lastly ran multiple times in a loop, until the minimum support was equal to zero.

* 1. Printing Lists to the Screen

For every minimum support value tested, the lists from L1 to Lk are printed to the console, so the actual output of the algorithm can be seen. It should be noted that the console output was added after the test runs, discussed in section 3 were completed. This is so the algorithm wouldn’t waste execution time by printing output to the screen, since console output is not related to the overall effectiveness of the algorithm.

1. Results

Figure 1 shows the results of running the algorithm being discussed in this paper on the T10.I4D.100K data set. To measure the effectiveness of this algorithm, the run time was measured in seconds, so that is how all the future graphs will be presented. It should be noted that in Apriori.py, the string separator used in the create\_transaction\_database function, on line 14, is the space character.

Figure 1

A close up of a map

Description automatically generatedFigure 2 shows the T10I4D100K data set run on the algorithm from the 1995 paper. This will allow for a comparison between this implementation and the one being evaluated in this paper.

Figure

Figure 3 shows the output of the algorithm being discussed in this paper on a random data set of 1000 transactions. The specific file used for the following runs was 1000-out1No\_Commas.txt. This size data set should be much more useful in determining the effectiveness of my algorithm because it can run much faster. It should be noted that in Apriori.py, the string separator used in the create\_transaction\_database function, on line 14, is the tab character.

Figure 3

Figure 4 shows the results of running the same 1000 transactions as stated previously from the implementation found here: https://github.com/omjego/AR-Mining-Hash-Tree.The file that was run was naive\_apriori.py. The parameters were minimum support as shown on the graph, and minimum confidence was set to zero since confidence was not measured in this paper’s is implementation. This implementation features Apriori, in Python, without a hash tree.

Figure 4

The data for figure 5 was obtained by running apriori\_with\_htree.py, from the link mentioned for figure 4. The parameters were minimum support as shown on the graph, and minimum confidence was set to zero since confidence was not measured in this paper’s is implementation. This implementation features Apriori, in Python, with a hash tree.

Figure 5

1. Discussion
   1. Reasons for using Python

Python is used to implement this algorithm for many reasons. Firstly, Python contains a built-in data type known as a dictionary. The dictionary is essentially Python’s version of a hash table. This allowed the implementation to be simpler because a hash tree no longer had to be implemented to store candidate itemsets. There were not many resources found for hash tree is related to data mining, so this was deemed the best viable option at the time.

Another reason for implementing Python is its ease of typecasting and dynamic typing. Being able to implement the Apriori algorithm without stringent data typing was very useful when trying to determine whether one itemset was a subset of a list. Python’s built-in subset function only works on sets, so quite often itemsets would have to be appended to a list and then converted to a set because sets are immutable in Python.

* 1. Comparison with 1995 Implementation

At first, a comparison between figure 1 and figure 2 looks very promising, but the algorithm being discussed in this paper (figure 1) has a timescale of hundreds of seconds, whereas figure 2’s timescale is only in seconds.

One thing that could be causing this issue is the difference between operating systems, and the languages that were chosen to implement the algorithm. Even though the current system has much faster hardware, Windows is a tendency to be substantially slower than UNIX. Similarly, many people were probably not using Python in 1995. The more likely option for the 1995 implementation is the C language. C is many times faster than Python, so that it is likely that the reason for the speed issues.

Current System

System: HP, 64-bit

Operating System: Windows

Processor: Intel Core i7, 3.1 GHz

RAM: 16 GB

Language Implementation: Python

System from 1995

System: IBM

Operating System: AIX 3.2 (UNIX)

Processor: 0.033 GHz

RAM: 0.064 GB

Language implementation: Probably not Python

Another reason could just be overall optimization issues in this new implementation. As stated in section 4.1, the reasons why Python was chosen to implement this algorithm can also lead to its optimization problems. Having to do so much typecasting, and not implementing an actual hash tree also causes algorithm to be much slower.

Another optimization issue is the use of many nested loops throughout the code in Apriori.py. This is especially true when having to iterate through the whole entire data set of 100,000 transactions.

* 1. Comparison with Python Implementations
     1. Modifications and Notes

It should be noted that in order to get the implementations from figures 4 and 5 to work correctly, modifications to the method that read in the data file were made, although they were very similar to the input methods in Apriori.py, from the implementation used to create the data in figure 3. Modifications were also made to the application’s main code, in order to test for multiple minimum support values, like in Driver.py, also from the implementation used to create the data in figure 3. Neither of these modifications changed how this person’s implementation of the actual Apriori algorithm runs. The modifications were done in order to easily graph the execution of the algorithm and input data.

It should also be noted that these algorithms were written in Python 2 and executed with a Python 2 compiler, which could also cause timing discrepancies, depending on the differences between Python 2 and 3.

* + 1. Comparison with the Python 2 Hash Tree Implementation

The algorithm corresponding to figure 5 allows for much more positive outcome of this paper’s implementation of the Apriori algorithm. This time the algorithm is only about 10 times slower than the one that is being compared to it.

As stated in sections 4.1 and 4.2, overall optimization issues and the reasons for choosing Python because the timing discrepancies between the two data sets. Figure 5 is implementation does use a hash table, which would lead one to believe that it is faster than the implementation presented in figure 3. Another reason could be that the implementation that was used to create the data in figure 5 was Python 2, as discussed in section 4.3.1.

* + 1. Comparison with the Python 2 Non-Hash Tree Implementation

The comparison between figure 3 and figure 4 is very promising. One will notice that the execution time of figure 3 is faster than that of figure 4 until the 0.25 minimum support level is reached.

At this point it is difficult to determine what caused the 0.25 support level to be so drastically different from the trends in figure 4. The code from the outside source is not written in a very meaningful way, nor is it commented very well.

One reason could be that the implementation that was used to create the data in figure 4 was Python 2, as discussed in section 4.3.1. Another reason is this person’s implementation might not support a minimum support value that is less than 1%. However, this person’s Hash Tree implementation, from figure 5, seems to work correctly for minimum support values that are less than 1%. Since both of those implementations were written by the same person, for comparative purppses, one would assume that they support the same range of possible minimum support values.

* 1. Overall Effectiveness of Algorithm

Generally speaking, it would appear that the algorithm presented in this paper performed very well compared to its more equivalent Python counterparts. That would lead one to believe that this algorithm is implemented correctly. It is actually known that the algorithm was implemented correctly because was initially tested on a very small data set that could be manually processed. However, the more important distinction is that timewise the algorithm tends to keep up with similar implementations. There will always be obvious exceptions due to differences in computer systems and levels of optimization, so it is believed that this algorithm performs decently for an implementation of Apriori.