Sign up Sign In

Search Medium

Write

Published in Towards Data Science

Chonyy Follow

Oct 26, 2020 · 7 min read · Listen

Apriori — Association Rule Mining In-depth Explanation and Python Implementation

Short and clear introduction to entry-level data mining.

462 | 4

Photo by Franki Chamaki on Unsplash

Introduction

The most famous story about association rule mining is the **"beer and diaper".** Researchers discovered that customers who buy diapers also tend to

buy beer. This classic example shows that there might be many interesting association rules hidden in our daily data.

Association rule mining is a technique to identify underlying relations between different items. There are many methods to perform association rule mining. The Apriori algorithm that we are going to introduce in this article is the most simple and straightforward approach. However, since it's the fundamental method, there are many different improvements that can be applied to it.

We will not delve deep into these improvements. Instead, I will show the major **shortcomings** of Apriori in this story. And in the upcoming post, a more efficient **FP Growth** algorithm will be introduced. We will also compare the pros and cons of FP Growth and Apriori in the next post.

FP Growth: Frequent Pattern Generation in Data Mining with Python Implementation

In this article, an advanced method called the FP Growth algorithm will be revealed. We will walk through the whole...

towardsdatascience.com

Concepts of Apriori

Support

> *Fraction of transactions that contain an itemset.*

For example, the support of item I is defined as the number of transactions containing I divided by the total number of transactions.

$$support(I) = \frac{Number\ of\ transactions\ containing\ I}{Total\ number\ of\ transactions}$$

Image by Chonyy

Confidence

> *Measures how often items in Y appear in transactions that contain X*

Confidence is the likelihood that item Y is also bought if item X is bought. It's calculated as the number of transactions containing X and Y divided by the number of transactions containing X.

$$confidence(X \rightarrow Y) = \frac{Number\ of\ transactions\ containing\ X\ and\ Y}{Number\ of\ transactions\ containing\ X}$$

Image by Chonyy

Frequent Item Set

> *An itemset whose support is greater than or equal to a minSup threshold*

Frequent itemsets or also known as frequent pattern simply means all the itemsets that the support satisfies the minimum support threshold.

Apriori Algorithm

Feel free to check out the well-commented source code. It could really help to understand the whole algorithm.

chonyy/apriori_python

pip install apriori_python Then use it like Get a copy of this repo using git clone git clone...

github.com

The main idea of Apriori is

> All non-empty subsets of a frequent itemset must also be frequent.

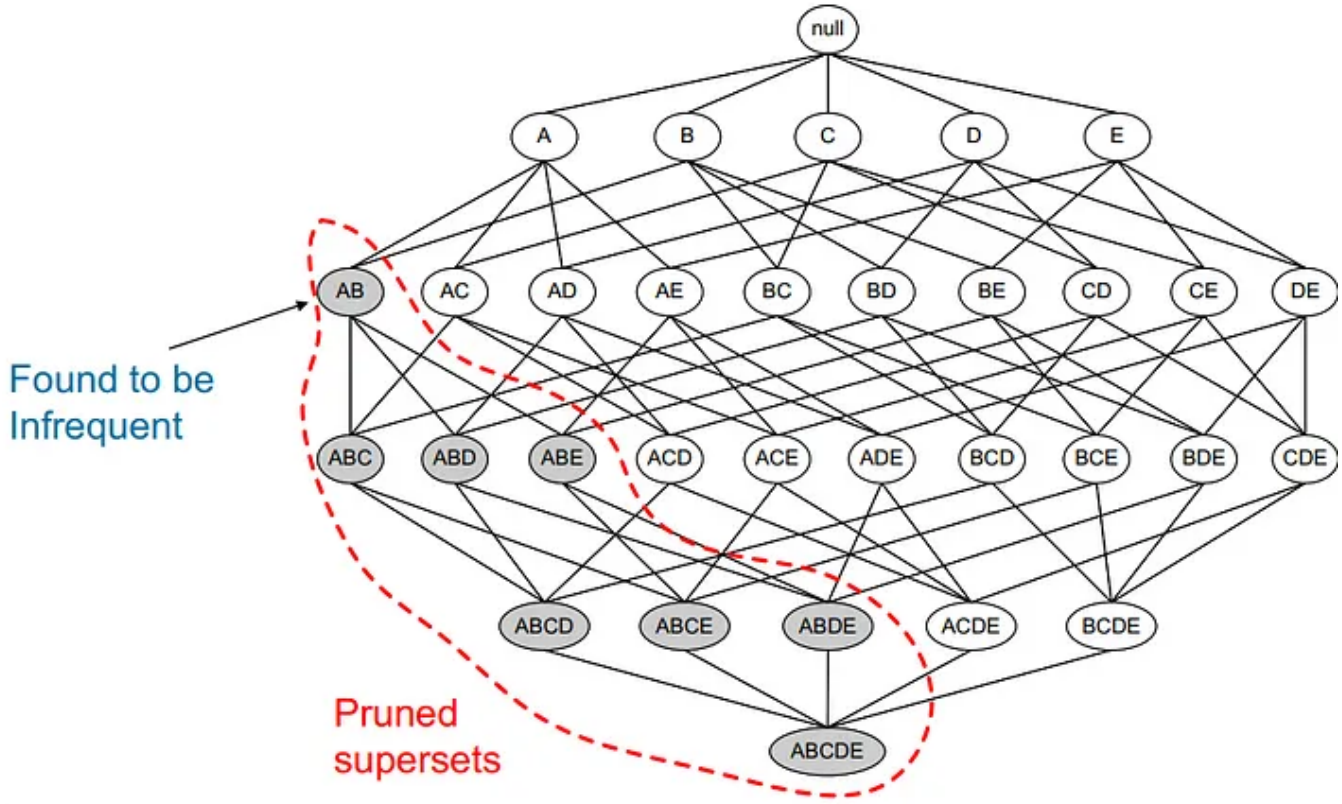

Image by Chonyy

It's a **bottom-up** approach. We started from every single item in the itemset list. Then, the candidates are generated by self-joining. We extend the length of the itemsets one item at a time. The subset test is performed at each stage and the itemsets that contain infrequent subsets are pruned. We repeat the process until no more successful itemsets can be derived from the data.

Algorithm Overview

This is the official pseudocode of Apriori

- **Lk:** frequent k-itemset, satisfy minimum support
- **Ck:** candidate k-itemset, possible frequent k-itemsets

L_1={frequent 1-itemsets};
for (k=2; $L_{k-1} \neq 0$; k++) do begin
    C_k=apriori-gen(L_{k-1});
    for each transactions t $\in$D do begin //scan DB
     C_t=subset(C_k, t) //get the subsets of t that are candidates
     for each candidate c $\in C_t$ do
          c.count++;
    end
    L_k={c $\in C_k$ | c.count $\geq$ minsup}
end
Answer=$\cup_k L_k$;

Image by Chonyy

Please be aware that the pruning step is already included in the apriori-gen function. Personally, I found this pseudocode quite confusing. So, I organized it into my own version. It should be way easier to understand.

```
L[1] = {frequent 1-itemsets};
for (k=2; L[k-1] != 0; k ++) do begin
    // perform self-joining
    C[k] = getUnion(L[k-1])
    // remove pruned supersets
```

```
    C[k] = pruning(C[k])
    // get itemsets that satisfy minSup
    L[k] = getAboveMinSup(C[k], minSup)
end
Answer = Lk (union)
```

To sum up, the basic components of Apriori can be written as

- Use k-1 itemsets to generate k itemsets
- Getting C[k] by joining L[k-1] and L[k-1]
- Prune C[k] with subset testing
- Generate L[k] by extracting the itemsets in C[k] that satisfy minSup

Simulate the algorithm in your head and validate it with the example below. The concept should be really clear now.

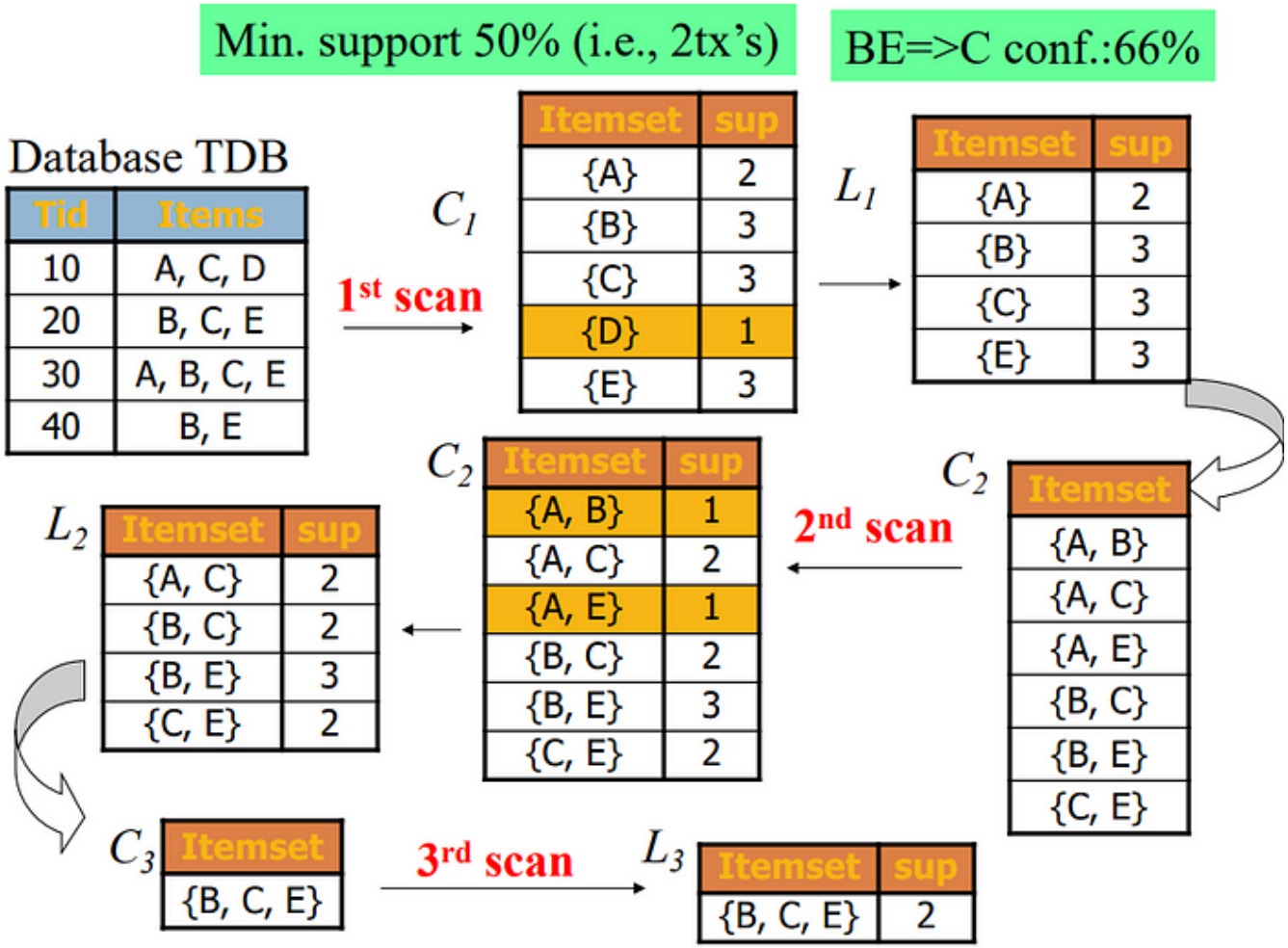

Image by Chonyy

Python Implementation

Apriori Function

This is the main function of this Apriori Python implementation. The most important part of this function is from **line 16 ~ line 21**. It basically follows my modified pseudocode written above.

1. Generate the candidate set by joining the frequent itemset from the previous stage.
2. Perform subset testing and prune the candidate set if there's an infrequent itemset contained.
3. Calculate the final frequent itemset by getting those satisfy minimum support.

```python
def apriori(itemSetList, minSup, minConf):
    C1ItemSet = getItemSetFromList(itemSetList)
    # Final result, global frequent itemset
    globalFreqItemSet = dict()
    # Storing global itemset with support count
    globalItemSetWithSup = defaultdict(int)

    L1ItemSet = getAboveMinSup(C1ItemSet, itemSetList, minSup, globalItemSetWithSup)
    currentLSet = L1ItemSet
    k = 2

    # Calculating frequent item set
    while(currentLSet):
        # Storing frequent itemset
        globalFreqItemSet[k-1] = currentLSet
        # Self-joining Lk
        candidateSet = getUnion(currentLSet, k)
        # Perform subset testing and remove pruned supersets
        candidateSet = pruning(candidateSet, currentLSet, k-1)
        # Scanning itemSet for counting support
        currentLSet = getAboveMinSup(candidateSet, itemSetList, minSup, globalItemSetWit
        k += 1

    rules = associationRule(globalFreqItemSet, globalItemSetWithSup, minConf)
    rules.sort(key=lambda x: x[2])

    return globalFreqItemSet, rules
```

apriori.py hosted with ❤️ by GitHub view raw

Candidate Generation

For self-joining, we simply get all the union through brute-force and only return those are in the specific length.

```python
def getUnion(itemSet, length):
    return set([i.union(j) for i in itemSet for j in itemSet if len(i.union(j)) == length
```

getUnion.py hosted with ❤ by GitHub view raw

Pruning

To perform subset testing, we loop through all possible subsets in the itemset. If the subset is not in the previous frequent itemset, we prune it.

```python
def pruning(candidateSet, prevFreqSet, length):
    tempCandidateSet = candidateSet.copy()
    for item in candidateSet:
        subsets = combinations(item, length)
        for subset in subsets:
            # if the subset is not in previous K-frequent get, then remove the set
            if(frozenset(subset) not in prevFreqSet):
                tempCandidateSet.remove(item)
                break
    return tempCandidateSet
```

pruning.py hosted with ❤ by GitHub view raw

Get Frequent Itemset from Candidate

In the final step, we **turn the candidate sets into frequent itemsets.** Since we are not applying any improvement technique. The only approach we can go for is to brainlessly loop through the item and itemset over and over again to obtain the count. At last, we only retain the itemsets whose support is equal or higher than minimum support.

```python
def getAboveMinSup(itemSet, itemSetList, minSup, globalItemSetWithSup):
    freqItemSet = set()
    localItemSetWithSup = defaultdict(int)

    for item in itemSet:
        for itemSet in itemSetList:
            if item.issubset(itemSet):
                globalItemSetWithSup[item] += 1
                localItemSetWithSup[item] += 1

    for item, supCount in localItemSetWithSup.items():
        support = float(supCount / len(itemSetList))
        if(support >= minSup):
            freqItemSet.add(item)

    return freqItemSet
```

getAboveMinSup.py hosted with ❤ by GitHub view raw

Result

```
print(rules)
# [[{'beer'}, {'rice'}, 0.666], [{'rice'}, {'beer'}, 1.000]]
# (rules[0] --> rules[1]), confidence = rules[2]
```

For more usage and examples, please check out the ***GitHub repo*** or the ***PyPi package***.

Shortcomings

There are two major shortcomings of Apriori Algorithms

- The size of itemset from candidate generation could be extremely large
- Lots of time wasted on counting the support since we have to scan the itemset database over and over again

We will use the data4.csv(generated from IBM generator) in the repo to showcase these shortcomings and see if we can get some interesting observations.

Candidate itemsets size at each stage

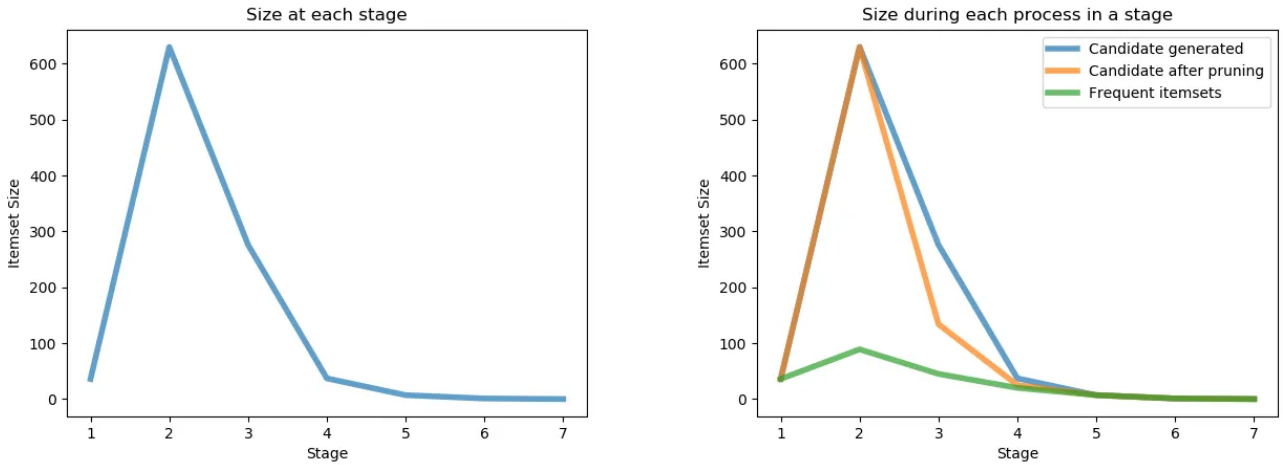

Image by Chonyy

By running Apriori on data4.csv, we can plot the process like the graph above. The shortcomings we mentioned above can be found in the observation of the graphs.

On the right, we can see the itemset size after the three major processes of the algorithm. Two key points can be discovered from the graph

- Size of itemset rapidly increase at the beginning, and gradually decrease as the iteration goes on
- Pruning process may be useless like stage 1 and 2. However, it could help a lot at some cases like stage 3. Half of the itemset is pruned, which means the counting time could be decreased by half!

Time elapsed at each stage

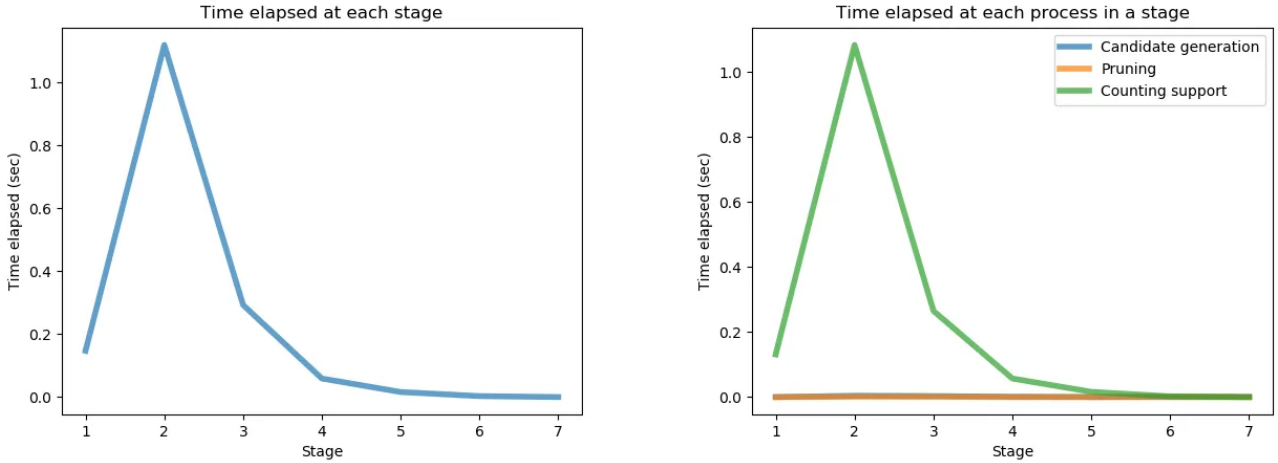

Image by Chonyy

From the plot, we can tell that most of the running is spent on counting the support. The time spent on candidate generation and pruning is nothing comparing to scanning the original itemset database over and over again.

Another observation worth attention is that we get a peak in cost at **stage 2.** Interestingly, This is actually not an accident! Data scientists often meet a bottleneck at stage 2 when using Apriori. Since there are almost no candidates removed at stage 1, the candidates generated at stage 2 are

basically all possible combinations of all 1-frequent itemsets. And calculating the support of such a huge itemset leads to extremely high costs.

Try on different datasets (in repo)

kaggle.csv

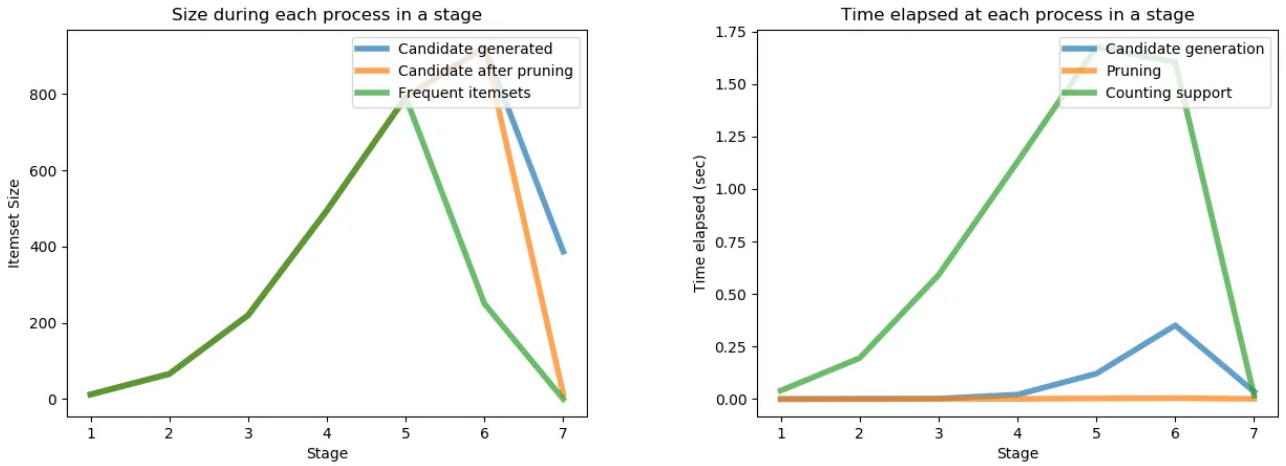

Image by Chonyy

Just like what we mentioned above, we knew that the bottleneck of Apriori is normally at stage 2. However, as it shows on the Kaggle dataset plot, this observation may not always hold. To be accurate, it depends on the dataset itself and the minimum support we want.

data7.csv

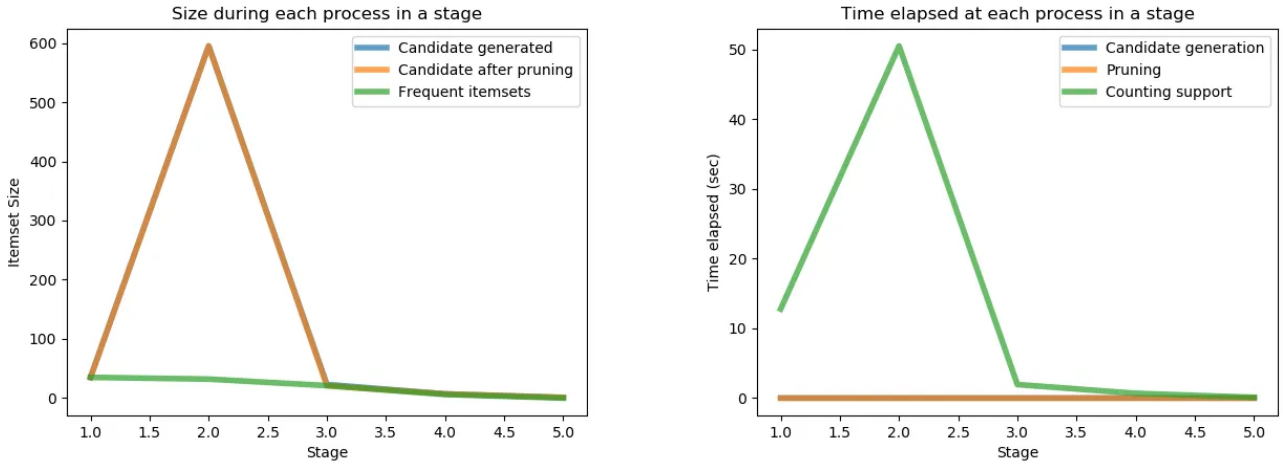

Image by Chonyy

As we can see, we need more than one minute to calculate the association rule of data7 with Apriori. Obviously, this running time is hardly acceptable. Remember that I said Apriori is just a fundamental method? The efficiency of it is the reason why it's not widely used in the data science field. We will take this result and compare it with the result from FP Growth.

FP Growth: Frequent Pattern Generation in Data Mining with Python Implementation

In this article, an advanced method called the FP Growth algorithm will be revealed. We will walk through the whole...

towardsdatascience.com

Improvements

There are many extra techniques that can be applied to Apriori to improve efficiency. Some of them are listed below.

- **Hashing:** reduce database scans
- **Transaction reduction:** remove infrequent transactions from further consideration
- **Partitioning:** possibly frequent must be frequent in one of the partition
- **Dynamic Itemset Counting:** reduce the number of passes over the data
- **Sampling:** pick up random samples

Source Code

chonyy/apriori_python

pip install apriori_python Then use it like Get a copy of this repo using git clone git clone...

github.com

PyPi Package

apriori-python

pip install apriori_python Then use it like To run the program with dataset provided and default values for minSupport...

pypi.org

Data Science Data Mining Fp Growth Machine Learning Python

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. Take a look.

By signing up, you will create a Medium account if you don't already have one. Review our Privacy Policy for more information about our privacy practices.

Get this newsletter

About Help Terms Privacy