# Frequent Pattern Mining on Adult Census Dataset

Tushar Kumar
University of Rochester
Rochester, New York
tusharku@UR.Rochester.edu

## ABSTRACT

This paper provides a detailed analysis of applying frequent pattern mining algorithms(Apriori, FPGrowth) on the Adult Census Dataset present in UCI ML repository. We talk about applying the two mining algorithms mentioned above on the dataset. We also improve the basic Apriori algorithm with the goal to improve its efficiency. We also extract association rules matching the input confidence threshold. We provide predictions on the test dataset using the association rules generated and Naive Bayes Algorithm. This analysis was done as part of the graduate course, CSC440 taken at University of Rochester.

## KEYWORDS

Pattern Mining, Apriori, FPGrowth, Association Rules, Naive Bayes, Adult Census Dataset

**ACM Reference Format:**
Tushar Kumar. 2018. Frequent Pattern Mining on Adult Census Dataset. Rochester, NY, USA, 8 pages.

## 1 INTRODUCTION

The task of pattern mining is extracting patterns that are prevalent in the dataset through usage of algorithms designed with the purpose of achieving that goal. Frequent pattern mining involves mining only those patterns that are *frequent* in the dataset.

The decision of when to actually acknowledge a recurring pattern as frequent is reached upon by basing it on a metric called *support*. Support of Support can be looked upon as the frequency of occurrence of a particular item in the dataset. Quite often we state support as a relative quantity calculated relative to the size of the dataset. Its computed as $N/D$ where N is the total number of records where that item is present and D is the total number of records in the database. Up until now, we have only been talking in terms of an item but we can instead also have *itemset* which will just be group of items and their support will follow the exact same definition as that of a singular item. The task of pattern mining then funnels down to attempting to find itemsets which occur frequently within the dataset. Using these frequently occurring itemsets one can find out rules which will enable us to derive important conclusions from the dataset. These rules, termed as association rules are of the form $A => B$ which means that presence of A in a record also implies presence of B in that record. These kind of rules allow us to solve some very interesting problems. A department store can derive what items are frequently bought together can place them nearby to increase the chances of customers buying both of them.

*CSC440 Fall'18, , Rochester, NY, USA*


One can also use these association rules for prediction. For example - if some attribute value implies the presence of target label that we are interested in, then we can predict the target class for any tuple where that attribute value is present. One might ask though, how do we believe that the rules are indeed "good". The metric that answers that question is *confidence* which indicates how strongly we can believe that rule to apply. Its range is $[0, 1]$ and is computed for an association rule. For a given association rule $A => B$, its computed as $\frac{Support(A \cup B)}{Support(A)}$ where $A \cup B$ refers to the transactions where both A and B occur.

I apply the pattern mining approach to Adult Census dataset[1]. I implement the following techniques and apply them on the dataset and analyze the results:

(1) *Implement Apriori algorithm and apply it on the dataset*
(2) *Implement FPGrowth algorithm and apply it on the dataset.*
(3) *Implement an improvement of the Apriori algorithm and apply it on the dataset*
(4) (For Extra Credit) *Discover Association rules passing an input minimum confidence level*
(5) (For Extra Credit) *Predict the "Salary" target label for a row using those association rules.*
(6) (For Extra Credit) *Implement the Naive Bayes Algorithm and predict the salary on test dataset.*

## 2 MINING ON ADULT CENSUS DATASET

Adult Census Dataset also known as "Census Income" dataset consists of attributes which are believed to have a relationship with their income. The goal of this dataset is to be able to build models, by training on the dataset, which have the ability to predict whether a person is earning more than 50K or less than 50K income. We first briefly define the properties of dataset in general before talking about each attributes:

(1) Dataset consists of 48842 instances, mix of continuous and discrete.
(2) To facilitate learning, the dataset is divided into trainig and test using a 66% and 33% split (train=32561, test=16281)
(3) Existence of imbalance. Percentage of rows with '>50K' : 23.93% and Percentage of rows with label '<=50K' : 76.07%
(4) Attribute values which are missing are replaced by "?"
(5) Contains a mixture of continuous and discrete attribute.

### 2.1 Attributes and their Properties

The Dataset consists of 15 attributes(including the salary attribute) of which 6 are continuous and rest all are discrete. Discrete attributes are the following:

[1] http://archive.ics.uci.edu/ml/datasets/Adult

(1) workclass - Describes the category and class of employment for the person
(2) education - Highest level of education the person has received
(3) marital status
(4) occupation
(5) relationship - Provides details on persons family and their role in it
(6) race - Person's ethnicity
(7) sex
(8) native country - Provides demographic details of the person
(9) salary - <=50K or >50K

Continuous attributes are the following:

(1) age
(2) fnlwgt - Estimate of the units of population that row represents.
(3) education-num - Number of years of education received.
(4) capital-gain
(5) capital-loss
(6) hours-per-week - Hours per week the person works

*2.1.1* ***Filtering attributes :*** The first task that I do in pre-processing this data is remove the fnlwgt attribute. The reason being that it does not provide any details on the chances of a person having more income than 50K. It only describes the weight that censor takers estimate that row to have. While we could weight each observation by that number and consider that to be our dataset but for the purpose of this analysis I have chosen to remove this attribute as otherwise it would unnecessarily complicate the calculation of support and candidates during mining.

*2.1.2* ***Handling Missing attributes :*** Quite a few tuples have attributes which instead of having a value, have a "?". This basically means the attribute value is unknown. While one could remove these instances completely from dataset with the goal to have a clean dataset, I chose to keep them. I replace each "?" with another constant "<attr_name>_unknown". Since the attributes which have a missing value are *workclass, occupation, native-country*, all of which are discrete, we can replace them with a constant. So if workclass attribute was missing we would instead replace it with *workclass_unknown* and likewise for others. The reason for doing this is that while that row has missing values, it can still act as support for a frequent itemset using the attribute values that are known. However we do not use these missing values for association rule generation and even for Naive Bayes algorithm. For those two implementations we completely ignore missing values.

*2.1.3* ***Discretization of continuous attributes :*** In order to discretize continuous attributes we look at the statistics of each attribute. Specially its mean and variance. Table 1 shows those statistic for each attribute. I have also plotted the distribution of these attributes in Figure 1. One can clearly see that while age,education-num and hours per week have a decent distribution but capital gain and loss are very very skewed towards zero and their variance is huge. So instead of finding equal sized bins for these two attributes, we divide them into three bins each{Zero, Low, High}. Zero corresponds to all zero values, Low corresponds to all non zero values which are less than the median of non zero values(7298 for Capital gain, 1887 for Capital loss). Do note that the medians are not medians of entire dataset but only median of non zero values of those attributes in dataset. This is done to get rid of the skewing effect on statistics of those attribute.
Age is discretized by dividing into bins of width 10 {10-19, 20-29, ...}. I divide education-num into bin size of 4{1-4, 5-8, ...} and hours per week into bins of size 20 {1-20, 21-40, ...}.

**Table 1: Mean and Variance of Continuous Attributes**

| Attribute Name | Mean | Variance |
|---|---|---|
| Age | 38.58163 | 186.06139 |
| Education-Num | 10.08058 | 6.61862 |
| Capital-Gain | 1077.61517 | 54542502.261 |
| Capital-Loss | 87.3065110 | 162376.7037 |
| Hours per Week | 40.437469 | 152.45898 |

*2.1.4* ***Converting all discrete values to integral representations :*** As part of pre-processing I also convert ALL attribute values to integral values through usage. In my implementation in Java, I converted all attribute values to Enums which is backed by an integer representation. The reason this is done is because it will improve the speed of computation by a huge margin. One of the things that we need to check while mining frequent item sets is presence of an item in a list(say while counting support of an itemset, you need to check if the current row has those values or not). This computation will be much more efficient if the list is a list of integers rather than a list of strings. This is why we convert all our attribute values to integers. Since we have already discretized all continuous attributes, this is easy to do as there are only a finite possible integral representations that we need to create(138 in my Enum representation). To summarize these are the ways I process

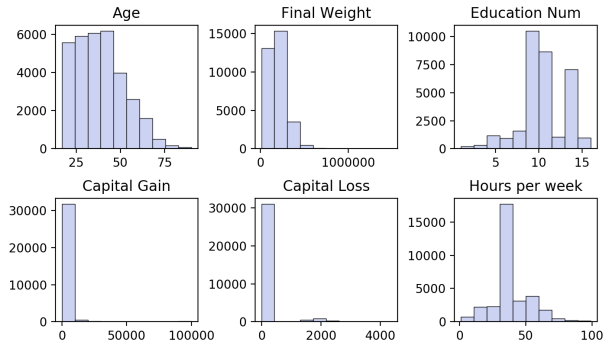


**Figure 1: Histogram of continuous attributes in dataset**

the data and transform it:

(1) Remove fnlwgt attribute
(2) Discretize age into bins of width 10
(3) Discretize education-num into bins of width 4
(4) Discretize hours-per-week into bins of width 20
(5) Discretize capital-gain into { Zero, Low(0 < gain ≤ MedianNonZero), High(MedianNonZero < gain < Max(gain)) }

(6) Discretize capital-loss into { Zero, Low(0 < loss ≤ MedianNonZero), High(MedianNonZero < loss < Max(loss)) }

Here MedianNonZero refers to the median of all the non zero values of that attribute.

## 2.2 Apriori

Apriori algorithm in data mining is used for mining frequent itemsets and relevant association rules. The key idea in Apriori algorithm is the anti-monotonicity of the support measure. It assumes that (i) All subsets of a frequent itemset must be frequent Similarly, for any infrequent itemset (ii) all its supersets must be infrequent too. With these two rules in place, Apriori algorithm is summarized in Figure 2 [1]. We first iterate over the entire dataset to get singular frequent items. Once we have that we combine[2] every pair of single item sets to generate candidates of frequent itemsets of size 2. This process is repeated till we keep on getting frequent itemset.

Whether an item set is frequent or not is decided in two steps. (i) We first filter and keep only those candidates of which every subset is a frequent item. This ensures that costly operation of checking whether an itemset is frequent or not by scanning dataset is kept to minimum cost. (ii) Once we have the filtered candidates, we then find the frequency of each candidate by iterating over all rows in dataset. We then generate only candidates which pass the minimum support criterion. Figure 3 shows the plot of number of itemsets generated for different input minimum support values. Given that the data is imbalanced, and heavily skewed towards "<=50K" label, we will be using a minimum support of 0.23 to extract itemsets with ">50K" also in them. Running Apriori with a minimum support of 0.23 generates the the output provided in Figure 4.

**Here are the key results of running Apriori on the Adult Census Dataset with a minimum support of 0.23:**

(1) Total number of transactions in Adult.data = 32561
(2) Total number of transactions scanned by Apriori = 260488 This makes sense since it generated frequent itemsets of size 7. So it must have scanned the dataset 8 times(and would not have found any frequent itemsets of size 8) . 260488 = 8 * 32561.
(3) Time taken to generate the itemsets = 1.48 seconds
(4) Total number of itemsets generated = 662
(5) Maximum itemset size = 7

## 2.3 FPGrowth

Apriori algorithm has two major disadvantages: (i) It generates huge number of candidates (ii) It requires as many scans of database as the size of largest frequent itemset.

The FP-Growth Algorithm is an efficient and scalable method for mining the complete set of frequent patterns by pattern fragment growth, using an extended prefix-tree structure for storing compressed and crucial information about frequent patterns named frequent-pattern tree (FP-tree). It overcomes the disadvantages of Apriori by straight away generating frequent itemsets themselves(and not potential candidates). It also only needs two scans of database no matter what the maximum itemset present. I have provided the pseudocode of FPGrowth in Figure 5 [1]

**Algorithm: Apriori.** Find frequent itemsets using an iterative level-wise approach based on candidate generation.

**Input:**

- $D$, a database of transactions;
- *min_sup*, the minimum support count threshold.

**Output:** $L$, frequent itemsets in $D$.

**Method:**

```
(1)   L1 = find_frequent_1-itemsets(D);
(2)   for (k = 2; Lk−1 ≠ φ; k++) {
(3)       Ck = apriori_gen(Lk−1);
(4)       for each transaction t ∈ D { // scan D for counts
(5)           Ct = subset(Ck, t); // get the subsets of t that are candidates
(6)           for each candidate c ∈ Ct
(7)               c.count++;
(8)       }
(9)       Lk = {c ∈ Ck | c.count ≥ min_sup}
(10)  }
(11)  return L = ∪k Lk;

procedure apriori_gen(Lk−1:frequent (k − 1)-itemsets)
(1)   for each itemset l1 ∈ Lk−1
(2)       for each itemset l2 ∈ Lk−1
(3)           if (l1[1] = l2[1]) ∧ (l1[2] = l2[2])
                  ∧... ∧ (l1[k − 2] = l2[k − 2]) ∧ (l1[k − 1] < l2[k − 1]) then {
(4)               c = l1 ⋈ l2; // join step: generate candidates
(5)               if has_infrequent_subset(c, Lk−1) then
(6)                   delete c; // prune step: remove unfruitful candidate
(7)               else add c to Ck;
(8)           }
(9)   return Ck;

procedure has_infrequent_subset(c: candidate k-itemset;
          Lk−1: frequent (k − 1)-itemsets); // use prior knowledge
(1)   for each (k − 1)-subset s of c
(2)       if s ∉ Lk−1 then
(3)           return TRUE;
(4)   return FALSE;
```

**Figure 2: Apriori Algorithm**

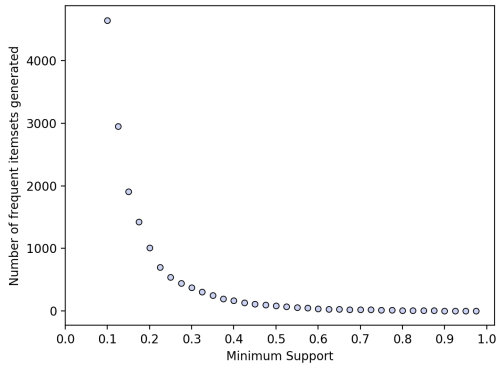


**Figure 3: Itemsets at different minimum support values with Apriori**

The basic idea is that we first scan the entire dataset and create a tree(FPTree) representation of the entire dataset. FPTree is created by sorting all the transactions in dataset in the decreasing order of the support counts of the singular items in the dataset. These are the only two scans needed. First scan is to find the frequency of every singular itemset. And second scan is to sort the transactions

```
Total Transactions : 32561

Total Transactions Scanned :260488

Time taken (in seconds) to find frequent itemsets with support of 0.23 = 1.461 seconds
==================================
Item Sets
==================================
------------------------------------------
Number of Itemsets of size 1 : 20
------------------------------------------
Printing one of them because of their huge number

{Age:20-29}
Relative Support : 0.24735112
------------------------------------------
Number of Itemsets of size 2 : 84
------------------------------------------
Printing one of them because of their huge number

{Relationship:Husband, Education-num:9-12}
Relative Support : 0.23736987
------------------------------------------
Number of Itemsets of size 3 : 172
------------------------------------------
Printing one of them because of their huge number

{Education-num:9-12, Race:White, Salary:Less than 50K}
Relative Support : 0.4259083
------------------------------------------
Number of Itemsets of size 4 : 207
------------------------------------------
Printing one of them because of their huge number

{Relationship:Husband, Education-num:9-12, Marital-status:Married-civ-spouse, Sex:Male}
Relative Support : 0.23712416
------------------------------------------
Number of Itemsets of size 5 : 131
------------------------------------------
Printing one of them because of their huge number

{Education-num:9-12, Race:White, Salary:Less than 50K, Workclass:Private,
Native-country:United-States}
Relative Support : 0.2936949
------------------------------------------
Number of Itemsets of size 6 : 42
------------------------------------------
Printing one of them because of their huge number

{Capital-gain:Zero, Capital-loss: Zero, Race:White, Salary:Less than 50K,
Hours-per-week:21-40, Native-country:United-States}
Relative Support : 0.34013084
------------------------------------------
Number of Itemsets of size 7 : 6
------------------------------------------
Printing one of them because of their huge number

{Capital-gain:Zero, Capital-loss: Zero, Education-num:9-12, Race:White,
Salary:Less than 50K, Workclass:Private, Native-country:United-States}
Relative Support : 0.27284175

Total Itemsets : 662
```

**Figure 4: Output of Apriori Algorithm at min_support = 0.23**

in the decreasing order of frequency counts and simultaneously add them to the FPTree. To add a sorted transaction to the tree we just follow through the nodes of the tree. If the first item in transaction is present as a child in the root then we move to that child, increase its support count by 1 and then process from the second item in transaction with new root being the child we just increased the support of. This is done recursively till entire that transaction is processed. This process is repeated for all transactions. We also maintain a header table which basically links all nodes which have the same value.

Once this FPTree is built we then start mining this tree to get frequent items. For each item in the header table(we iterate in increasing order of support), we build a conditional fp tree. Which is just the tree of nodes which have that item as the leaf node. We then recursively call FPGrowth algorithm on this conditional fp tree. There are two places we generate frequent item sets. (i) Whenever we reach a point where the tree(conditional fp tree) only has one path, we can generate all itemsets we obtain by different combinations[2] of nodes in that tree as frequent itemsets. (ii) While creating the conditional FPtree, we generate itemsets created by union of the current FP tree's header table items and the previous trees(in recursion stack) conditional pattern base. For the first time, it would just be the header table values and from where we will get the singular frequent items.

**Algorithm**: **FP_growth.** Mine frequent itemsets using an FP-tree by pattern fragment growth.

**Input**:

- $D$, a transaction database;
- *min_sup*, the minimum support count threshold.

**Output**: The complete set of frequent patterns.

**Method**:

1. The FP-tree is constructed in the following steps:
   - (**a**) Scan the transaction database $D$ once. Collect $F$, the set of frequent items, and their support counts. Sort $F$ in support count descending order as $L$, the *list* of frequent items.
   - (**b**) Create the root of an FP-tree, and label it as "null." For each transaction *Trans* in $D$ do the following.
     Select and sort the frequent items in *Trans* according to the order of $L$. Let the sorted frequent item list in *Trans* be $[p|P]$, where $p$ is the first element and $P$ is the remaining list. Call insert_tree($[p|P]$, $T$), which is performed as follows. If $T$ has a child $N$ such that $N.item\text{-}name = p.item\text{-}name$, then increment $N$'s count by 1; else create a new node $N$, and let its count be 1, its parent link be linked to $T$, and its node-link to the nodes with the same *item-name* via the node-link structure. If $P$ is nonempty, call insert_tree($P$, $N$) recursively.
2. The FP-tree is mined by calling FP_growth(*FP_tree*, *null*), which is implemented as follows.

```
procedure FP_growth(Tree, α)
(1)  if Tree contains a single path P then
(2)    for each combination (denoted as β) of the nodes in the path P
(3)      generate pattern β ∪ α with support_count = minimum support count of nodes in β;
(4)  else for each a_i in the header of Tree {
(5)    generate pattern β = a_i ∪ α with support_count = a_i.support_count;
(6)    construct β's conditional pattern base and then β's conditional FP_tree Tree_β;
(7)    if Tree_β ≠ ∅ then
(8)      call FP_growth(Tree_β, β); }
```

**Figure 5: FPGrowth Algorithm**

Figure 6 shows the plot of number of itemsets generated for different input minimum support values woth FPGrowth. As expected this should be *exactly* the same as FPGrowth output. The verbose output of FPGrowth for the minimum support of 0.23 further confirms the equivalence of itemsets generated. Not that the one sample that I print out is chosen randomly so it might be different but the number of itemsets and the set of itemsets are exactly the same. **The key observations of the output are:**

(1) Total number of transactions in Adult.data = 32561
(2) Total number of transactions scanned by FPGrowth = 65,122 This makes sense since it only scans the dataset two times. 65122 = 2 * 32561.
(3) Time taken to generate the itemsets = 0.268 seconds This is 5.5X faster than Apriori for generating the itemsets on Adult.data.
(4) Total number of itemsets generated = 662 (Same as Apriori)
(5) Maximum itemset size = 7 (Same as Apriori)

The only two differences between Apriori and FPGrowth are that (i) it generates the itemsets much quickly and (ii) it scans the dataset only two times.

## 2.4 Improving Apriori

Even though Apriori does the job for us, we would still want to improve its efficiency. For a dataset of just 32561 rows, it took 1.5

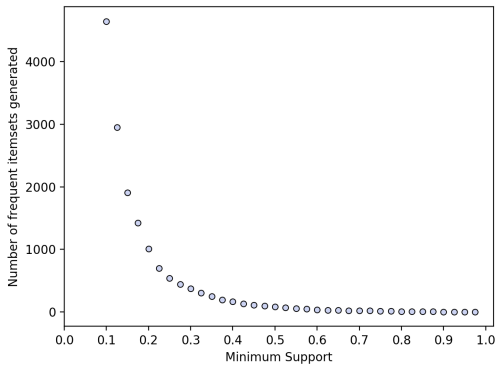


**Figure 6: Itemsets at different minimum support values with FPGrowth**

```
Total Transactions : 32561

Total Transactions Scanned :65122

Time taken (in seconds) to find frequent itemsets with support of 0.23 = 0.268 seconds
==================================
Item Sets
==================================
------------------------------------------
Number of Itemsets of size 1 : 20
------------------------------------------
Printing one of them because of their huge number

{Age:20-29}
Relative Support : 0.24735112
------------------------------------------
Number of Itemsets of size 2 : 84
------------------------------------------
Printing one of them because of their huge number

{Salary:Less than 50K, Age:20-29}
Relative Support : 0.23171893
------------------------------------------
Number of Itemsets of size 3 : 172
------------------------------------------
Printing one of them because of their huge number

{Capital-gain:Zero, Native-country:United-States, Education:HS-grad}
Relative Support : 0.2785234
------------------------------------------
Number of Itemsets of size 4 : 207
------------------------------------------
Printing one of them because of their huge number

{Capital-loss: Zero, Capital-gain:Zero, Education-num:9-12, Hours-per-week:21-40}
Relative Support : 0.36513007
------------------------------------------
Number of Itemsets of size 5 : 131
------------------------------------------
Printing one of them because of their huge number

{Native-country:United-States, Race:White, Workclass:Private, Sex:Male, Education-num:9-12}
Relative Support : 0.23604926
------------------------------------------
Number of Itemsets of size 6 : 42
------------------------------------------
Printing one of them because of their huge number

{Sex:Male, Marital-status:Married-civ-spouse, Capital-loss: Zero, Race:White,
Native-country:United-States, Relationship:Husband}
Relative Support : 0.3164829
------------------------------------------
Number of Itemsets of size 7 : 6
------------------------------------------
Printing one of them because of their huge number

{Capital-loss: Zero, Capital-gain:Zero, Native-country:United-States, Race:White,
Salary:Less than 50K, Workclass:Private, Sex:Male}
Relative Support : 0.23414515

Total Itemsets : 662
```

**Figure 7: Output of FPGrowth Algorithm at min_support = 0.23**

seconds. This would increase to a really huge value had the dataset been of 10 times huge. In fact I simulated datasets of different sizes by duplicating each row N times to see the impact on run times of algorithms. Figure 8 shows the comparison of FPGrowth and Apriori with different scales of dataset size where a dataset of scale say 500 was created by duplicating each record in adult.data 500 times. This would result into a dataset of size 16,280,500 = 16 Million Rows. But this does not mean we should just throwaway Apriori.

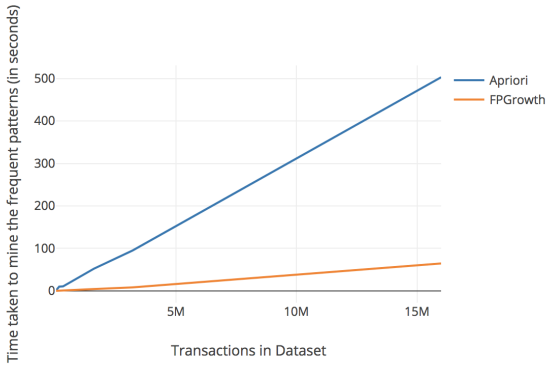


**Figure 8: Comparison of FPGrowth and Apriori based on Dataset Size.**

There can be certain improvements that can be made to improve the efficiency of Apriori. In particular I implemented two:

(1) ***Parallelization of itemset frequency check***. One of costly operation that Apriori does is checking for all the candidate itemsets if it meets the minimum support criteria. Taking inspiration from the text book's[1, p. 255] idea of partitioning dataset, I decided to do the same during support check scan. I divide the dataset into N equal partitions. And run the scan of each partition parallely in different threads wherein I iterate over all the candidates and keep a count of number of times that itemset appears in that partition. I have a concurrent map where each thread keeps on increasing the count of the itemset. Having a concurrent map ensures the udpates are done in a thread safe manner. Once all threads are complete we have the count of all candidates in the entire dataset. And then we can remove all candidates which do not meet minimum threshold criteria. In my implementation I use N = 5, but for larger datasets we should have many more partition. Specifically when I simulated for dataset of size 16M, I kept N as 200.

(2) ***Reduce the number of transactions scanned.*** Another improvement that I added was that while I am finding the support of candidates for kthitemset, I remove all rows from dataset which do not have any frequent k item set. The reason being that if they don't have a kth frequent itemset then they definitely cannot have a k+1 frequent itemset and will be of no further use. If you are using the partitioned approach then its easy to just update each partition in their own thread by removing rows which do not generate any frequent item and then once all threads are complete, just combine the partitions in the order you split them to create the new dataset.

```
/**
 * Have concurrent map to ensure that all
 * updates to support of an item set
 * are done in a thread safe manner.
 */
Map<Integer, Integer> itemSetToFrequencyMapping = new ConcurrentHashMap<Integer, Integer>();
for(int j = 0; j < kthItemSetCandidatesAfterPruning.size(); j++){
    itemSetToFrequencyMapping.put(j, 0);
}
List<List<Integer>> kthItemSet = new ArrayList<List<Integer>>();

/**
 * Spawn a thread so that each thread will handle each partition
 * Giving it a value of 5 should be fine as that would not be causing
 * too much load and not end up doing more harm than good.
 * Not using fixed thread pool because I am anyways creating smaller
 * number of threads so we can afford to start a new one rather
 * than waiting for existing ones to be idle.
 */
ExecutorService executorService = Executors.newCachedThreadPool();
List<Future<?>> tasks = new ArrayList<Future<?>>();
for(List<Row> rows : partitionedRows) {
    tasks.add(executorService.submit(new Runnable() {
        /**
         * Runnable to extract the support for each candidate
         * in each partition in parallel and update the count
         * in the concurrent map created above.
         */
        @Override
        public void run() {
            findSupportForEachCandidateInEachParition(kthItemSetCandidatesAfterPruning,
                    itemSetToFrequencyMapping, rows);
        }
    }));
}
/**
 * Block for all five partition scans to be complete
 */
for(Future<?> task : tasks) {
    try {
        task.get(1, TimeUnit.MINUTES);
    } catch (InterruptedException | ExecutionException | TimeoutException e) {
        /**
         * One of the tasks could not be completed within the specified
         * time limits of 60 seconds.
         */
        System.err.println("One of the future tasks could not be completed "
                + "within specified timeout");
        e.printStackTrace();
    }
}
```

**Figure 9: Code to parallelize the support check of entire dataset.**

With these changes, we see improvement in the timings of Apriori algorithm. These improvements are put behind another algorithm which is referred as Apriori-Improved henceforth. Figure 10 shows the comparison of Apriori, Apriori-Improved and FPGrowth. As one can see the changes significantly improve the efficiency of Apriori even on much larger dataset sizes. However for very large dataset sizes, FPGrowth is still performing better than both Apriori and Apriori-Improved. I also run Apriori-Improved on the adult

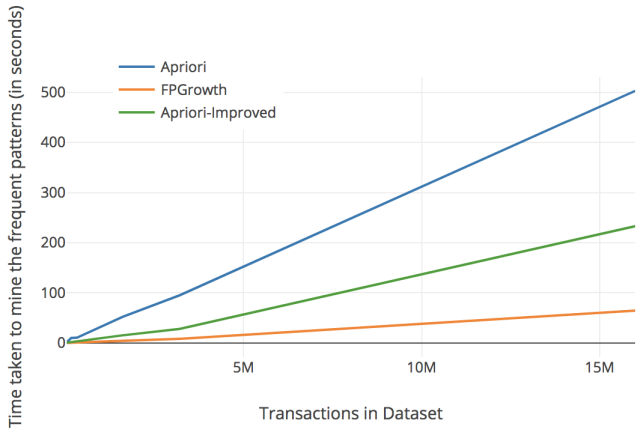


**Figure 10: Comparison of Apriori, Apriori-Improved and FP-Growth based on Dataset Size.**

dataset with the minimum support of 0.23. Figure 11 provides the verbose output of that run. There are couple of observations which

```
Total Transactions : 32561

Total Transactions Scanned :247024

Time taken (in seconds) to find frequent itemsets with support of 0.23 = 0.654 seconds
==================================
Item Sets
==================================
------------------------------------------
Number of Itemsets of size 1 : 20
------------------------------------------
Printing one of them because of their huge number

{Age:20-29}
Relative Support : 0.24735112
------------------------------------------
Number of Itemsets of size 2 : 84
------------------------------------------
Printing one of them because of their huge number

{Relationship:Husband, Education-num:9-12}
Relative Support : 0.23736987
------------------------------------------
Number of Itemsets of size 3 : 172
------------------------------------------
Printing one of them because of their huge number

{Education-num:9-12, Race:White, Salary:Less than 50K}
Relative Support : 0.4259083
------------------------------------------
Number of Itemsets of size 4 : 207
------------------------------------------
Printing one of them because of their huge number

{Relationship:Husband, Education-num:9-12, Marital-status:Married-civ-spouse, Sex:Male}
Relative Support : 0.23712416
------------------------------------------
Number of Itemsets of size 5 : 131
------------------------------------------
Printing one of them because of their huge number

{Education-num:9-12, Race:White, Salary:Less than 50K, Workclass:Private,
Native-country:United-States}
Relative Support : 0.2936949
------------------------------------------
Number of Itemsets of size 6 : 42
------------------------------------------
Printing one of them because of their huge number

{Capital-gain:Zero, Capital-loss: Zero, Race:White, Salary:Less than 50K,
Hours-per-week:21-40, Native-country:United-States}
Relative Support : 0.34013084
------------------------------------------
Number of Itemsets of size 7 : 6
------------------------------------------
Printing one of them because of their huge number

{Capital-gain:Zero, Capital-loss: Zero, Education-num:9-12, Race:White,
Salary:Less than 50K, Workclass:Private, Native-country:United-States}
Relative Support : 0.27284175

Total Itemsets : 662
```

**Figure 11: Output of Apriori-Improved at min_support of 0.23**

we should focus on:

(1) Reduced number of transactions scanned. Apriori-Improved was able to get rid of 13,464 transaction scans.
(2) Apriori Improved is much more efficient. It took only 0.6 seconds to mine the datset whereas Apriori took 1.5 seconds.
(3) Its no surprise that it also generates the same number of frequent itemsets. Considering all we are changing is adding parallelization.

## 2.5 Association Rule Generation and Prediction

In order to perform prediction using the frequent patterns generated, I have implemented mechanism of generating association rules from the patterns mined, which are more than the input confidence. In order to generate the association rules, I just gather the frequent patterns and for each frequent itemset do the following:

(1) Generate two partitions of the itemset $s$ which we call $l$ and $s - l$ such that $l$ contains only the target class label (<=50K or >50K) .

(2) Generate all rules which have $Confidence(l, l-s) > min\_conf$ where $Confidence(A, B) = \frac{Support(A \cup B)}{Support(B)}$

In order to generate association rules which also have >50K labels in them, I give a very low support value of 0.02. That is, a support requirement of only 2% of the transactions. However, I ensure that only rules with strong believability are generated by keeping a high confidence requirement of 0.99 for the adult dataset. These minimum support and minimum confidence values help us to get a lot of rules for >50K also, which is great because its the class thats under-represented in our dataset and they also give the best result in terms of all performance metrics(accuracy,precision,recall and f1 score). Figure 12 illustrates some of the association rules generated with the above values of support and confidence.

```
============================
Sample Association Rules Extracted for Less than 50K

------2 Sample Association Rules------

{ Capital-loss: Zero, Capital-gain:Zero, Race:White, Education-num:9-12,
 Marital-status:Never-married, Occupation:Adm-clerical } => Less than 50K
Confidence = 0.99234134


{ Race:White, Sex:Female, Marital-status:Never-married,
Education:HS-grad } => Less than 50K
Confidence = 0.9934569


============================

Sample Association Rules Extracted for More than 50K

------2 Sample Association Rules------

{ Native-country:United-States, Sex:Male, Marital-status:Married-civ-spouse
, Capital-gain:High (7299-99999) } => More than 50K
Confidence = 0.9911635


{ Marital-status:Married-civ-spouse, Native-country:United-States,
Relationship:Husband, Capital-gain:High (7299-99999) } => More than 50K
Confidence = 0.99115044


============================

Running Predictions using the Association Rules Generated On Test Data

------------------------------------------------------------
Precision : 0.989517819706499
Recall : 0.7827529021558872
F1 Score : 0.8740740740740741
Negative Predictive Value : 0.9769447377683914
Accuracy of Predictions on test data using association rules: 0.9779184932618932
```

**Figure 12: Output of association rule generation and prediction on test data**

*2.5.1* ***Prediction Performance :*** For prediction, my strategy is simple. With all the association rules generated on the training dataset, we iterate over the test data set and do the following operations for each tuple in test data:

(1) If there is only one rule whose precedent is present in the row then we predict the consequent of this rule as the target label of this test row.
(2) If there are multiple rules whose precedent is present in the row then we choose the rule which has higher confidence value and predict the consequent of that rule as the target label of this test row.

We evaluate our predictions on all four measures. With the value of min_support = 0.02 and confidence = 0.99, we get the following numbers:
*Accuracy : 0.977918*
*Precision : 0.989517*
*Recall : 0.782752*
*F1 Score : 0.87407*
*Negative Predictive Value : 0.97694*
We get very high positive and negative predictive value.

## 2.6 Naive Bayes

I also test prediction on test data using Naive Bayes algorithm. In order to perform prediction using Naive Bayes we do not ignore the missing values. Instead what we do is add a constant for that. For any attribute index 'i' whose value is missing and has a "?" in the dataset, we replace it with "<i>_Unknown". This will enable us to build probabilities for the missing rows also and will also allow us to predict it when we get a row with "?" in test data. though this leads to a lesser number of rows this will enable us to build correct probabilities. In order to predict on test data we do the following:

(1) Build conditional probability distribution for all attributes for both the classes. This will enable us to compute $P(attributeValue|Class)$.
(2) For discrete attribute this is simply obtained by the count of occurrences of rows of that class which have that discrete value.
(3) For continuous this is obtained by assuming a Gaussian distribution for that attribute for all values of that attribute for that class. With that assumption, once we know the mean and variance of that attribute for that class, its easy to compute the probability using the formula for Gaussian distribution.

$$P(X_{continuous}|Class_i) = \frac{1}{\sqrt[2]{2\pi\sigma^2}} \exp^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

```
==================================
Running Naive Bayes Algorithm
==================================
Gathering Data to build class Probabilities
Building Probability Tables
Gathering predictions based on probabilities


==================================
Performance metrics for Naive Bayes
==================================
Treating <= 50K as negative class and >50K as positive class
True Positive : 2300.0
False Positive : 1163.0
True Negative : 11272.0
False Negative : 1546.0
---------------------------
Precision : 0.6641640196361536
Recall : 0.5980239209568383
F1 Score : 0.6293610617047476
Negative Predictive Value : 0.8793883601185832
Accuracy on test data using Naive Bayes: 0.8336097291321172
```

**Figure 13: Output of Naive Bayes prediction on test data**

*2.6.1* ***Prediction Performance :*** For prediction of test data, we simply use the probabilities built to figure out which class has higher probability by using Bayes theorem and assumption of conditional independence of attribute values given the class label. Figure 13 provides the output of running Naive Bayes on test data after extracting the probabilities from training data.

Even though the accuracy is decent enough by comparing the results mentioned on the adult census data repository, its still low.

Few reasons I feel this is the case is because of the asusmption of gaussian distribution for all continuous attributes, specially capital gain and loss, which are definitely not gaussian. Another reason why this is the case is because of under representation of positive data. Only 23% of the entire dataset and hence we just could not gather enough data to correctly identify positive examples, which is why our negative predictive value is high but our positive predictive value is low. I evaluate our predictions on all four measures and get the following results:
*Accuracy : 0.833609*
*Precision : 0.664164*
*Recall : 0.598023*
*F1 Score : 0.62936*
*Negative Predictive Value : 0.87938*
We get good negative(<=50K) predictive value but low positive(>50K) predictive value

## 3 CONCLUSIONS

I implemented Apriori , FpGrowth and an improved version of Apriori and analyze the result frequent itemset generation on Adult Census Dataset. I conclude that FPGrowth is the fastest algorithm to generate frequent itemsets. Though Apriori can be improved by making changes targeting its efficiency, its still cannot outperform FPGrowth. I also found that association rule generation whilst being a simple algorithm still ends up giving us good results in terms of prediction. Naive Bayes on the other hand wasn't able to give us the same performance in terms of accuracy but thats primarily because of the class imbalance. Association rules could get rid of that challenge by having a very low support as its constraint.

## ACKNOWLEDGMENTS

I would like to thank Professor Ted Pawlicki for giving me this opportunity to implement these algorithms and analyze them which has fostered a much deeper understanding of these methods. This report was generated as part of the mini project requirement given to us for a graduate course in Data mining at University of Rochester taught by Professor Ted Pawlicki

## REFERENCES

[1] Jian Pei Jiawei Han, Micheline Kamber. 1993. *Data Mining: Concepts and Techniques (3rd Ed.).* Morgan Kaufmann; 3 edition (July 6, 2011).
[2] theproductiveprogrammer. 2018. How to Generate Combinations. https://theproductiveprogrammer.blog/GeneratingCombinations.java.php