COMS 4995: Parpriori Report

Claire Liu (cl3944), Evan Li (el3078)

December 22, 2022

1 Introduction

Our project parellelizes the apriori algorithm for association rule mining in Haskell. Association rule mining is commonly used in the commerce space, where shoppers purchase a certain set of items in each transaction, to determine which sets of items are frequently purchased together. Association rule mining has typically been used to reveal surprising trends hidden in data. A more general problem statement can be found in the paper Fast Algorithms for Mining Association Rules by Agrawal and Srikant. We primarily run our experiments on a dataset of 180k restaurant health ratings in New York City. Each transaction in this dataset represents a anonymized restaurant in NYC and includes a borough, type of cuisine, and health inspection grade.

2 Sequential Apriori Algorithm

2.1 Overview

The apriori algorithm uses a bottom up approach to first find all large 1-item sets that satisfy a minimum support. Then, via a candidate generation algorithm, the large 2-item sets, 3-item sets, and so on are formed. As we form candidate sets, we also apply a pruning algorithm that removes all candidates that don't meet our minimum support. Once, we've finished generating frequent itemsets, we can form our strong association rules. After generating the association rules, we keep the ones that fulfill our minimum confidence. Here is what our restaurant health rating input file looks like:

```
$ head data/restaurants.csv
     BROOKLYN , Chinese , Z
     MANHATTAN , American , C
     BRONX , Chicken , C
     MANHATTAN , American , A
     MANHATTAN , American , A
     MANHATTAN , American , C
     BRONX , Pizza , A
     MANHATTAN , Caribbean , B
     MANHATTAN , Italian , B
     STATENISLAND , Bakery , A
```

To run the sequential algorithm, use:

```
$ stack exec parpriori -- <filename> <min_support> <min_confidence> <seq/par>
$ stack exec parpriori -- data/restaurants.csv 0.004 0.55 seq
    [" A"," French "] [" MANHATTAN "] 0.809931506849315
    [" French "] [" MANHATTAN "] 0.7979990471653168
    [" C"," Italian "] [" MANHATTAN "] 0.6765650080256822
    [" Irish "] [" MANHATTAN "] 0.6671398154719659
    [" B"," Italian "] [" MANHATTAN "] 0.6383734701934465
    [" Korean "] [" QUEENS "] 0.627441161742614
    [" Italian "] [" MANHATTAN "] 0.6165966386554622
    [" Japanese "] [" MANHATTAN "] 0.6031938899496615
    [" B"," Japanese "] [" MANHATTAN "] 0.5842050209205021
    [" A"," Cafe/Coffee/Tea "] [" MANHATTAN "] 0.5840283042709122
    [" A"," Italian "] [" MANHATTAN "] 0.582916855974557
    [" A"," Japanese "] [" MANHATTAN "] 0.5739329268292683
    [" American "," C"] [" MANHATTAN "] 0.5565684518964675
```

Each line of output represents a derived association rule with its confidence score. For example, the most confident association rule is that a French restaurant with a health score of A will be in Manhattan with 81% confidence.

2.2 Sequential Candidate Generation

First, the code will generate large 1-item sets which consists of calculating the frequency of each item for every transaction and filtering out those that don't satisfy our minimum support. Then we pass the large (k-1) item sets into our candidate generation algorithm in order to produce the large k-item sets. The function performs a self join using list comprehension and runs in $O(n^2)$ time. As an example, the large 3-item sets: [1,2,3] [1,2,4] [1,2,5] would produce the following 4-item candidates: [1,2,3,4], [1,2,3,5], [1,2,4,5].

```
aprioriGen :: [([Text], Int)] -> [[Text]]
aprioriGen items = removeDups (Prelude.concat((Prelude.map (\x -> (createCand x prev_L_items))
prev_L_items)))
                    where prev_L_items = Prelude.map (\(x, _) -> x) items

createCand :: [Text] -> [[Text]] -> [[Text]]
createCand p prev_L_items = [L.sort (p ++ [(L.last q)]) | q <- prev_L_items, L.take (L.length p - 1) p
== L.take (L.length q - 1) q, (L.last p) /= (L.last q)]
```

2.3 Sequential Pruning

After we generate all item sets of some size k using our candidate generation algorithm, we only want to keep item sets that meet our minimum support threshold. We obtain the support count for an item set by looping through all of our transactions and counting how many times that item set is a subset of a transaction. Subset checking is done using standard Prelude list functions. If the support count divided by the total number of transactions is above our minsup thresh-hold, we keep this item set and add it to our collection of k-item sets that we will eventually use for our next run of candidate generation.

```
prune :: [[Text]] -> [[Text]] -> [([Text], Int)]
prune [] [] = []
prune citems baskets = Prelude.map (\item -> (item, supCount item baskets)) citems
```

2.4 Sequential Confidence Calculation of Association Rules

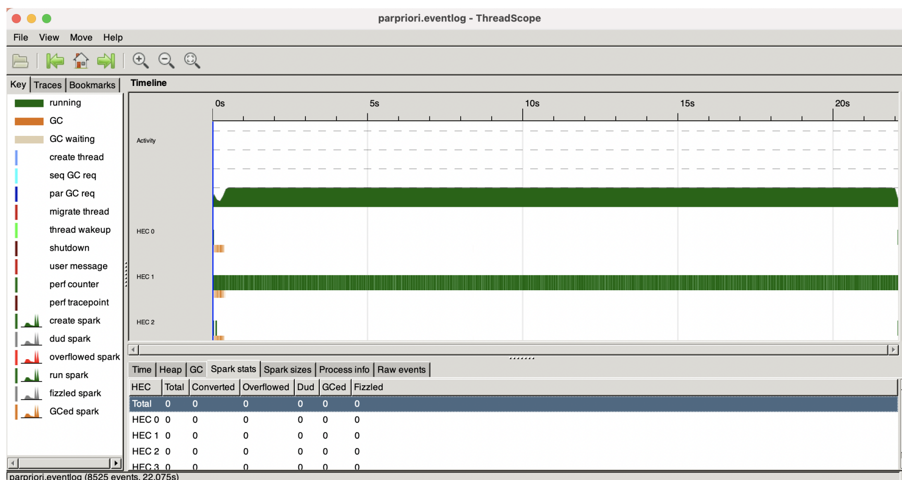
At this point, we have obtained all item sets and their corresponding support scores. Using this information, we create map that allows us to look up the support score of any item set. Then, for each item set, we generate all possible association rules from that item set. For example, given a 4-item set of 1, 2, 3, 4, we generate the associations 1→(2, 3, 4), (1, 2)→(3, 4), and (1, 2, 3)→4. We do not need to worry about rules such as (1, 2)→3 since those rules are processed when we are looping through 3-item sets. We then calculate the confidence score of each generated rule, and keep the rules with confidence scores (obtained using the formula conf(a → b) = sup(a u b)/sup(a)) above minconf.

```
getConfidence :: [[Text]] -> Map [Text] Double -> Double -> [([Text], [Text], Double)]
getConfidence [] _ _  = []
getConfidence xs support_map minconf = xs >>= (\x -> (getConfidence' x support_map minconf))

getConfidence' :: [Text] -> Map [Text] Double -> Double -> [([Text], [Text], Double)]
getConfidence' [] _ _ = []
getConfidence' [_] _ _ = []
getConfidence' (a:ab) support_map minconf
    | isNothing numerator = []
    | otherwise = extractCor [a] ab support_map (fromJust numerator) minconf
    where numerator = M.lookup (a:ab) support_map
```

2.5 Sequential Results

Running our sequential algorithm on one core, on a dataset of around 180K transactions took 22 seconds.



3 Parallel Apriori Algorithm

3.1 Parallel Candidate Generation

Previously, candidate generation happened sequentially so that the createCand function was mapped to every element of the list. However, it's not necessary for this mapping to be sequential so we used the parMap function to evaluate the resulting list in parallel using the createCand function.

```
aprioriGenPar :: [([Text], Int)] -> [[Text]]
aprioriGenPar items = removeDups (Prelude.concat((parMap rpar (\x -> (createCand x
prev_L_items)) prev_L_items)))
                    where prev_L_items = Prelude.map (\(x, _) -> x) items

createCand :: [Text] -> [[Text]] -> [[Text]]
createCand p prev_L_items = [L.sort (p ++ [(L.last q)]) | q <- prev_L_items, L.take
(L.length p - 1) p == L.take (L.length q - 1) q, (L.last p) /= (L.last q)]
```

3.2 Parallel Pruning

As described before, our pruning algorithm is a nested for loop where the outside loop iterates through each newly generated item set and the inside loop iterates over each transaction in our dataset. We parallelize the outside loop by using the parMap function to run a helper function supCount, which takes in an item set and list of transactions and returns the support count for that itemset, for each newly generated item set.

```
prunePar :: [[Text]] -> [[Text]] -> [([Text], Int)]
prunePar [] [] = []
prunePar citems baskets = parMap rpar (\item -> (item, supCount item baskets)) citems
```

3.3 Parallel Confidence Calculation of Association Rules

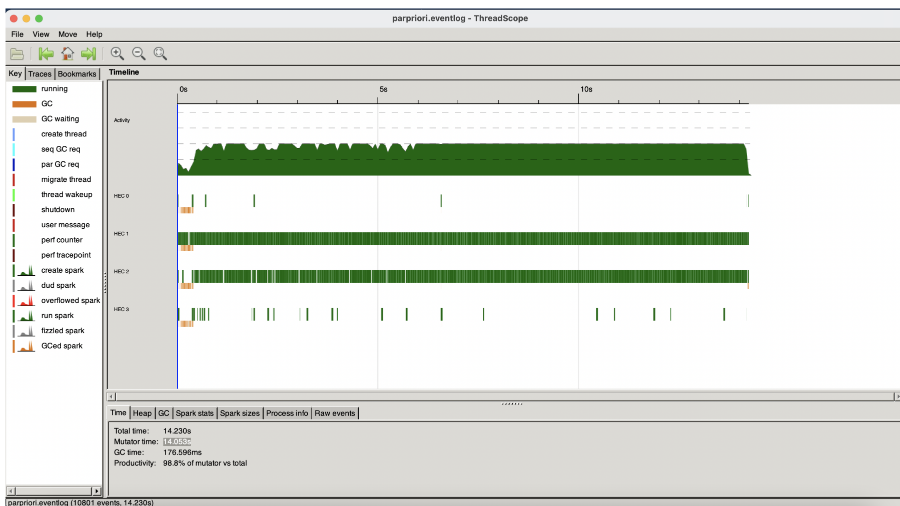
Our confidence calculations for association rules requires two main steps 1) for each item set, generate its association rules 2) find the confidence scores for the generated association rules by performing lookups on our support map. We parallelize the 'for each item set' component of these two steps, so that generating association rules and calculating confidence scores can be done in parallel for each item set. This was also accomplished using the parMap function.

```haskell
getConfidencePar :: [[Text]] -> Map [Text] Double -> Double -> [([Text], [Text], Double)]
getConfidencePar [] _ _  = []
getConfidencePar xs support_map minconf = Prelude.concat (parMap rpar (\x -> (getConfidence' x support_map
minconf)) xs)

getConfidence' :: [Text] -> Map [Text] Double -> Double -> [([Text], [Text], Double)]
getConfidence' [] _ _ = []
getConfidence' [_] _ _ = []
getConfidence' (a:ab) support_map minconf
    | isNothing numerator = []
    | otherwise = extractCor [a] ab support_map (fromJust numerator) minconf
    where numerator = M.lookup (a:ab) support_map
```

3.4 Parallel Results

After running it on the same dataset of 180K transactions, but using the parallel version with 4 cores, we see a speedup to 14 seconds for around 35% improvement. We also see two active threads coming from the parallelization of pruning and confidence calculations, indicating that these contributed to the speed up significantly.



4 Code Listings

4.1 Usage

Our submission is a Haskell project developed using stack. To run the parallel program run the following commands:

```
$ stack clean
$ stack build
$ stack exec parpriori -- data/restaurants.csv 0.004 0.5 par +RTS -ls -N4
```

Our sample data is included in the data directory for test runs.

4.2 main()

```haskell
main :: IO ()
main = do
      args <- getArgs
      case args of
         [filename, minsupS, minconfS, "par"] -> do
            contents <- readFile filename
            let items = preprocess (T.lines (strip $ T.pack contents))
                minsup = read minsupS :: Double
                minconf = read minconfS :: Double
                dataLen = (Prelude.length items)
                --loaded argument data
                wordLst = Prelude.concat items
                counts = M.toList (M.fromListWith (+) (Prelude.map (\x -> (x, 1)) wordLst))
                l1 = Prelude.map (\(a, b) -> ([a], b)) (Prelude.filter (\(_, cnt) -> (getSup cnt
dataLen) >= minsup) counts)
                --construct L1 itemset
                iSets = (itemSetsPar l1 items dataLen minsup)
                --obtained Lk itemsets
                support_map = (getSupportMap iSets dataLen)
                --generate support map
                correlations = sortedConfidence (getConfidencePar (Prelude.map (\(x, _) -> x) iSets)
(M.fromList support_map) minconf)
                --obtain correlations
            mapM_ (printCor) correlations
         [filename, minsupS, minconfS, "seq"] -> do
            contents <- readFile filename
            let items = preprocess (T.lines (strip $ T.pack contents))
                minsup = read minsupS :: Double
                minconf = read minconfS :: Double
                dataLen = (Prelude.length items)
                --loaded argument data
                wordLst = Prelude.concat items
                counts = M.toList (M.fromListWith (+) (Prelude.map (\x -> (x, 1)) wordLst))
                l1 = Prelude.map (\(a, b) -> ([a], b)) (Prelude.filter (\(_, cnt) -> (getSup cnt
dataLen) >= minsup) counts)
                --construct L1 itemset
                iSets = (itemSets l1 items dataLen minsup)
                --obtained Lk itemsets
                support_map = (getSupportMap iSets dataLen)
                --generate support map
                correlations = sortedConfidence (getConfidence (Prelude.map (\(x, _) -> x) iSets)
(M.fromList support_map) minconf)
                --obtain correlations
            mapM_ (printCor) correlations
         _ -> do
            pn <- getProgName
            die $ "Usage: "++pn++" <filename> <minsup> <minconf> <seq/par>"
```

4.3 Support Map

```haskell
---SUPPORT MAP CONSTRUCTION---
getSup :: Int -> Int -> Double
getSup cnt datalen = (fromIntegral cnt) / (fromIntegral datalen)

getSupportMap :: [([Text], Int)] -> Int -> [([Text], Double)]
getSupportMap isets datalen = Prelude.map (\(x, cnt) -> (x, (getSup cnt datalen)))
isets
```

4.4 Item Sets

```
---ITEM SET CONSTRUCTION---
itemSets :: [([Text], Int)] -> [[Text]] -> Int -> Double -> [([Text], Int)]
itemSets [] _ _ _ = []
itemSets prev_L_items items datalen minsup = prev_L_items ++ (itemSets l_items items datalen minsup)
    where
        c_items = aprioriGen prev_L_items
        l_items = Prelude.filter (\(_, cnt) -> (getSup cnt datalen) >= minsup) (prune (c_items) items)

itemSetsPar :: [([Text], Int)] -> [[Text]] -> Int -> Double -> [([Text], Int)]
itemSetsPar [] _ _ _ = []
itemSetsPar prev_L_items items datalen minsup = prev_L_items ++ (itemSetsPar l_items items datalen
minsup)
    where
        c_items = aprioriGenPar prev_L_items
        l_items = Prelude.filter (\(_, cnt) -> (getSup cnt datalen) >= minsup) (prunePar (c_items)
items)
```

4.5 Candidate Generation

```
--CANDIDATE GENERATION---

removeDups :: [[Text]] -> [[Text]]
removeDups xs = S.toList(S.fromList(xs))

aprioriGenPar :: [([Text], Int)] -> [[Text]]
aprioriGenPar items = removeDups (Prelude.concat((parMap rpar (\x -> (createCand x prev_L_items))
prev_L_items)))
                    where prev_L_items = Prelude.map (\(x, _) -> x) items

aprioriGen :: [([Text], Int)] -> [[Text]]
aprioriGen items = removeDups (Prelude.concat((Prelude.map (\x -> (createCand x prev_L_items))
prev_L_items)))
                    where prev_L_items = Prelude.map (\(x, _) -> x) items

createCand :: [Text] -> [[Text]] -> [[Text]]
createCand p prev_L_items = [L.sort (p ++ [(L.last q)]) | q <- prev_L_items, L.take (L.length p - 1) p
== L.take (L.length q - 1) q, (L.last p) /= (L.last q)]
```

4.6 Pruning

```
--PRUNING--
prune :: [[Text]] -> [[Text]] -> [([Text], Int)]
prune [] [] = []
prune citems baskets = Prelude.map (\item -> (item, supCount item baskets)) citems

prunePar :: [[Text]] -> [[Text]] -> [([Text], Int)]
prunePar [] [] = []
prunePar citems baskets = parMap rpar (\item -> (item, supCount item baskets))
citems
supCount :: [Text] -> [[Text]] -> Int
supCount _ [] = 0
supCount item (basket:xs)
    | (Prelude.all (`Prelude.elem` basket) item) = 1 + (supCount item xs)
    | otherwise = supCount item xs
```

4.7 Confidence Scoring

```
--CONFIDENCE RELATIONS MINING---

sortedConfidence :: [([Text], [Text], Double)] -> [([Text], [Text], Double)]
sortedConfidence xs = (sortBy (\(_, _, a) (_, _, b) -> compare b a) xs)

getConfidencePar :: [[Text]] -> Map [Text] Double -> Double -> [([Text], [Text], Double)]
getConfidencePar [] _ _  = []
getConfidencePar xs support_map minconf = Prelude.concat (parMap rpar (\x -> (getConfidence' x
support_map minconf)) xs)

getConfidence :: [[Text]] -> Map [Text] Double -> Double -> [([Text], [Text], Double)]
getConfidence [] _ _  = []
getConfidence xs support_map minconf = xs >>= (\x -> (getConfidence' x support_map minconf))

getConfidence' :: [Text] -> Map [Text] Double -> Double -> [([Text], [Text], Double)]
getConfidence' [] _ _ = []
getConfidence' [_] _ _ = []
getConfidence' (a:ab) support_map minconf
    | isNothing numerator = []
    | otherwise = extractCor [a] ab support_map (fromJust numerator) minconf
    where numerator = M.lookup (a:ab) support_map

extractCor :: [Text] -> [Text] -> Map [Text] Double -> Double -> Double -> [([Text], [Text], Double)]
extractCor _ [] _ _ _ = []
extractCor [] _ _ _ _ = []
extractCor lfs (r:rhs) support_map numerator minconf
    | isNothing lsup = (extractCor (lfs ++ [r]) rhs support_map numerator minconf)
    | conf >= minconf = (lfs, (r:rhs), conf) : (extractCor (lfs ++ [r]) rhs support_map numerator
minconf)
    | otherwise = (extractCor (lfs ++ [r]) rhs support_map numerator minconf)
    where
        lsup = M.lookup lfs support_map
        conf = (numerator / (fromJust lsup))
```

4.8 Helper I/O Functions

```
---HELPER DATA PREPROCESSING FNS---
printCor :: ([Text], [Text], Double) -> IO()
printCor (t1, t2, conf) = putStrLn ((show t1) ++ " " ++ (show t2) ++ " " ++ (show conf))

preprocess :: [Text] -> [[Text]]
preprocess [] = []
preprocess (x:xs) = (T.splitOn (T.pack ",") x) : (preprocess xs)
```