Title: Scalable Cluster-Based Analysis of Twitter Data

Author: Eric Shang-Ping Dai, DXXSHA001

Project Abbreviation: SLAMS

Supervisor(s): Jivashi Nagar, Prof. Hussein Suleman

<table>
<thead>
<tr>
<th>Category</th>
<th>Min</th>
<th>Max</th>
<th>Chosen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement Analysis and Design</td>
<td>0</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Theoretical Analysis</td>
<td>0</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>Experiment Design and Execution</td>
<td>0</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>System Development and Implementation</td>
<td>0</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>Results, Findings and Conclusion</td>
<td>10</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>Aim Formulation and Background Work</td>
<td>10</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>Quality of Paper Writing and Presentation</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Quality of Deliverables</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Overall General Project Evaluation (this section allowed only with motivation letter from supervisor)</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

Total marks 80 80
Scalable Analysis on a Cluster of Twitter Data Using Mining Techniques

Association Rule Mining Module

Shang-Ping Dai
Computer Science Honours
University of Cape Town
Cape Town, South Africa
dxxsha001@myuct.ac.za

ABSTRACT
Social media platforms generate large volumes of user generated data on a daily basis. This data can be used to find interesting patterns by extracting relationships between topics that are being discussed in social media platforms. Due to the issues faced when dealing with large data, a system is presented in this paper that can perform a scalable analysis on Twitter data. This paper focuses on the data mining module that extracts interesting relationships and patterns in the dataset. The main focus of the module is to deal with the scalability of the dataset size efficiently and to be able to compute and generate patterns and relationships from Twitter data. The resulting module implemented a parallel FP-Growth association rule mining algorithm on a Map-Reduce framework, which makes use of Amazon’s AWS EMR to mine a given set of Twitter data. The results show that the performance of the rule mining module is dependent on the specifications of the type of cluster used. The execution times of the module is seen to be able to scale slower with the dataset sizes if the cluster is given more nodes.

CCS Concepts
• Data mining → Association rule mining
• Information systems → Cloud-based storage

KEYWORDS
Association Rule Mining; Cluster Computation; Map-Reduce

1. INTRODUCTION
Large amounts of data are produced daily [10], especially in social media platforms. Social media platforms produce user-generated content, where users of the platform often vocalize and share opinions or thoughts on various topics. These topics can vary from being recent, personal events, or regarding the state of affairs of the world [9]. Twitter is one of the most popular social media platforms in use, where many of its users communicate their opinions online daily.

Valuable insights can be gained about online communities and individuals by analyzing such data. This can be done with the use of data mining. Data mining is the process of generating insightful information by examining (usually large) sets of data. Specifically, data mining can be used to find unexpected patterns within datasets [2]. The application of data mining in social media data can yield many benefits, such as gauging user sentiment or developing effective recommendation systems.

Various difficulties are faced when dealing with user-generated content. Often, social media posts can be non-sensical or incoherent, especially if no context is provided [9]. Most systems would have to preprocess or even prune such content if one wishes to analyze the data in a meaningful way. Another challenge arises when dealing with large amounts of data. A meaningful analysis of data does not scale well with large datasets [11].

The aim of this project is to develop a data analysis system for researchers to analyze large datasets. The system utilizes a cluster to perform a scalable analysis on large amounts of Twitter data. The system will consist of three modules: a data mining module that will perform parallel association rule mining to generate association rules on the Twitter data, a parallel information retrieval module that can retrieve a subset of the entire dataset for the association rule module to work on, and a Web-based front-end application which a user interacts with to request and configure jobs.

This paper focuses on the data mining module of the system. This module aims to perform scalable association rule mining on large datasets to generate results in a reasonable amount of time.

2. BACKGROUND & RELATED WORK

2.1 Association Rule Mining
Association rule mining aims to find interesting correlations and frequent patterns (usually) in transactional, market-based databases or other data repositories [2]. It does this by building a set of rules (usually very large) from the dataset. A rule relates a set of items, \( I_{left} \), with another set of items, \( I_{right} \). This relationship implies that if \( I_{left} \) appears in a transaction, there is a likely chance that \( I_{right} \) will also appear in that transaction. These sets of items are called itemsets. Rules are only established for popular itemsets, that is, for itemsets that occur frequently in the dataset. These are called frequent itemsets.
Association rule mining can be split into two sub problems [2]. The first is the generation of frequent itemsets; this step finds all itemsets whose occurrences exceed a given minimum support threshold. The second sub problem is the generation of association rules using the frequent itemsets. Rules are generated with a confidence threshold, which is the likelihood with which the itemsets in the rules appear in a transaction. It has been identified that the slowest performing step of association rule mining is the generation of frequent itemsets. Association rules are generated from frequent itemsets by taking each combination of frequent itemsets and calculating the confidence of the rule based on the support count of the itemset [16]. Progress on increasing the efficiency of rule mining algorithms were made by the parallelization of rule mining algorithms [12] and by reducing the number of passes over the dataset [13].

Two popular association rule mining algorithms: the standard Apriori algorithm and the Frequent-Pattern Growth algorithm will be discussed and explained in Section 2.1.1 and Section 2.1.2.

2.1.1 Standard Apriori Algorithm

The standard Apriori algorithm for association rule mining, as described by Agrawal et al [3], generates all frequent itemsets in the dataset by first determining all candidate itemsets. These itemsets are itemsets that we hope will be frequent. These candidate itemsets are then checked for their respective frequencies in the dataset and are then determined to be frequent or not when comparing to the support threshold. This algorithm uses the Apriori principle to find frequent itemsets. This principle states that if an item is infrequent, then any itemset containing that item will also be infrequent. To take advantage of this principle, on the first iteration, all frequent items are found. These are all items whose support count is above the support threshold. These items are then combined for every combination of candidate itemsets of size two. These candidate-2-itemsets are then checked with the dataset for their support count to find all frequent-2-itemsets and so on. This process is then repeated until all frequent itemsets are found and association rules are then generated with these itemsets.

This means the algorithm must scan through the dataset \( k \) times where \( k \) is the number of items in the frequent itemset with the highest number of items. A list of all combinations of frequent items would also need to be generated for each iteration, which makes this approach very inefficient for large amounts of data.

2.1.2 Frequent-Pattern Growth Algorithm

The Frequent-Pattern Growth algorithm (FP-Growth) [4] is one of the more efficient implementations for finding frequent itemsets in a dataset. The algorithm does this by constructing an FP tree and mining frequent itemsets from the tree. This is done by first finding all frequent items in the dataset, similar to the Apriori algorithm. These items are then sorted by frequency and the order is applied to each transaction. Infrequent items are also removed from each transaction. The FP tree is then constructed by adding each ordered transaction to the tree such that each transaction becomes a path from the root of the tree to a leaf node. This is done for all transactions. Transactions with similar paths will then increment a count for the nodes which it shares a path with. The final tree will then show all frequent items and which other items they appear with in the dataset and their respective support levels with those items. All this information is then used to mine frequent itemsets. To illustrate further, FP-Growth will be demonstrated on the list of transactions presented in Table 1.

<table>
<thead>
<tr>
<th>Transaction ID</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A, C, T, H, R, B</td>
</tr>
<tr>
<td>2</td>
<td>T, B, H, C</td>
</tr>
<tr>
<td>3</td>
<td>T, C, H</td>
</tr>
<tr>
<td>4</td>
<td>C, H, B</td>
</tr>
<tr>
<td>5</td>
<td>T, B</td>
</tr>
<tr>
<td>6</td>
<td>B, S, Q, H</td>
</tr>
</tbody>
</table>

The frequent items are first found in this list of transactions. The minimum support for this example will be 3. By scanning through the list once, the list of frequent items and their support count are: C: 4, T: 4, H: 5, B: 5. All transactions are then ordered by the ordering of the frequent pattern. The frequent pattern is the order of frequent items by their support count. In this example, the frequent pattern would be: B, H, C, T. The list of ordered transactions is then presented in Table 2.

<table>
<thead>
<tr>
<th>Transaction ID</th>
<th>Ordered Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B, H, C, T</td>
</tr>
<tr>
<td>2</td>
<td>B, H, C, T</td>
</tr>
<tr>
<td>3</td>
<td>H, C, T</td>
</tr>
<tr>
<td>4</td>
<td>B, H, C</td>
</tr>
<tr>
<td>5</td>
<td>B, T</td>
</tr>
<tr>
<td>6</td>
<td>B, H</td>
</tr>
</tbody>
</table>

Notice that all infrequent items are also ignored in every transaction.

The FP-tree is then constructed with the ordered transactions. This is done by adding each ordered transaction as paths from the root node of the tree to their respective leaf node. If paths are similar, they are joined together, and their counts are incremented. The resulting FP-tree for this example is shown in Figure 1.

Figure 1. FP-tree constructed from the ordered transactions list.
Each item, item, in the header table is now used to construct frequent itemsets that include that item. We do this by constructing the item’s conditional pattern base and thus finding the item’s conditional FP-tree and then mining all itemsets based on that. The conditional pattern base of item is found by finding all paths from the root node to the item. This is called the conditional pattern base since these transactions occur with the condition of item’s existence being in the same transaction. The conditional FP-tree of item then determines which items that occur with item are frequent. For example, for item C: its conditional pattern base will be the paths \([B, H]:3\) and \([H]:1\). This is found by traversing all paths from the root node to each C node (specified by all linked C nodes in the header table). These conditional pattern bases show the separate, unique instances of other items that appear with item C in the dataset. The counts used for each conditional pattern base is the cumulative count, \(c_{nodeC}\), of the node that is reached traversing this path since the items on this path only occur with that node \(c_{nodeC}\) times. The conditional FP-Tree for item C is then found using item C’s conditional pattern bases. The conditional FP-tree for item C would be \(<B:3>, <H:4>\). This can be confirmed from the list of transactions since C occurs with B three times and C occurs with H four times in the transactions in the dataset. The frequent itemsets that can be found with C would thus be all combinations of C with the items in C’s conditional FP-tree. These are: \{B, C\}, \{H, C\}, \{B, H, C\}.

FP Growth was found to perform much better than the Apriori algorithm because it only requires scanning through the dataset twice. The first scan finds the frequent items of the dataset, and the second scan is used to construct the FP tree from the ordered transactions.

2.1.3 Map-Reduce Framework

The Map-Reduce framework [5] is a programming interface that simplifies many algorithms if they can be implemented with a map and reduce method. This is highly applicable to data mining algorithms due to the simplicity of the framework’s design. The framework is seen to improve the performance of rule mining algorithms [6, 7, 8].

In brief, inputs to the program are partitioned into pieces and given to a map method. The map method maps the inputs to some key-value pair. All output key-value pairs are shuffled, and equal keys have their values aggregated to a list. Each unique key is then passed to a reduce method that reduces the list of values relating to a key to some output key-value pair. This output pair is then written as the output of the program. This process is illustrated in Figure 2.

A popular implementation of the Map-Reduce framework is Apache Hadoop [6, 7, 8]. Hadoop provides a Java implementation for mapping and reducing. Hadoop clusters allow for implementations to easily parallelize over multiple machines. Hadoop does this by internally load balancing and assigning map and reduce jobs to nodes in a native Hadoop cluster. Hadoop clusters consist of one master node and many worker nodes that work on Map-Reduce jobs. Hadoop also natively uses the Hadoop Distributed File System (HDFS) to store inputs and outputs of Map-Reduce jobs. This allows for Hadoop to recover failed map or reduce calls with its replication of data on the HDFS.

2.2 Social Media Mining Systems

Zhang et al [1] developed a pattern-based analysis system that analyzed Sina Weibo data, a Chinese social media platform similar to Twitter. Their developed system made use of a pipelined approach where the social media data was first preprocessed before mining for patterns in the data. This preprocessing was done to filter out unnecessary words in each post so that topic analysis and detection could be performed on the data. To identify popular patterns in the data, their mining algorithm was a parallel implementation of an FP-growth mining algorithm. They had implemented this on Spark, an interface that allows programs to run on Hadoop. Their system also included a web-based visualization of the processed data for user interaction and analysis. The system, however, processed social media data in real-time, rather than processing a finite dataset.

3. REQUIREMENTS ANALYSIS AND DESIGN

3.1 Approach

The data mining module focuses on the approach taken for the implementation of a scalable data mining algorithm that can mine patterns from provided Twitter data efficiently with increasing dataset sizes. The following sections will discuss the design of the data mining module within the data analysis system.

3.1.1 Requirements

The aim of the system is to aid researchers to analyze large amounts of data in a scalable manner. To this extent, in the initial phases of designing the system, requirements for the system were
gathered from PhD students in the field of Computer Science. Requirements were gathered in the form of a questionnaire. The questionnaire contained questions relating to expectations that users have for systems that process large amounts of data. Questions regarding the back-end specifically touched on what dataset sizes the user generally works on, expected computation times of those datasets and what types of processes are done on the datasets.

The following were the preliminary requirements for the back-end modules of the system before development.

- Provide pipelining capabilities for the staging, pre-processing and analysis of data.
- Expectations of queries and data mining jobs to complete in an order of minutes. Very dependent on job size.

3.1.2 Software Development Methodology

The software was developed using the Rapid Application Development (RAD) methodology. RAD is an iterative development methodology [14] that reduces risk of not meeting deadlines. RAD uses a development cycle in which development begins with requirements gathering to determine the requirements of the system. Software is then built with core functionality and demonstrated to stakeholders or users to gain feedback. The process is then repeated to refine the software and evaluation is done at the end of software development. This was advantageous to the development of the project since it was highly applicable to the deadlines of the project.

The data mining module followed an iterative approach where various iterations were developed and evaluated for feedback to tune and improve upon in future iterations. The rule mining module underwent three major iterations in the development process. Broadly, the first iteration implemented the standard Apriori algorithm for association rule mining, the second iteration implemented a semi-parallelized version of the FP-Growth algorithm for association rule mining and the third and final version implemented a fully-parallelized version of FP-Growth. These iterations are further detailed in Section 4.1.

3.2 Data Mining Module Design

3.2.1 Module Design Decisions

The association rule mining module was developed as a Map-Reduce Apache Hadoop program. To this extent, Java was used as the development language. Hadoop provides many advantages for the scalability of programs. This is due to the fact that Hadoop automatically load-balances Map and Reduce function calls to all machines in the cluster. Additionally, Hadoop automatically replicates and keeps track of all replications of data so that interrupted function calls can be handled without any errors. Another advantage that Hadoop offered was that the implemented algorithm would only have to be implemented as Map and Reduce methods to take advantage of Hadoop’s features.

The rule mining module was developed on a single-node local Hadoop set up. This version emulated worker nodes by running virtual worker nodes on the local machine.

It was decided that the rule mining module would implement the FP-Growth association rule mining algorithm, as it was one of the most efficient and scalable algorithms to mine rules with.

3.2.2 Final Implementation Design

The class diagram shown in Appendix A shows the classes used for the implementation of the FP-Growth algorithm.

The FPgrowthMain class is the main class that runs all Map-Reduce jobs. Nested in FPgrowthMain are three Mapper and Reducer pairs for each of the Map-Reduce jobs that are run in the main method. These pairs consist of the ItemsMapper and ItemsReducer classes, the OrderedItemsetMapper and FreqItemsetsReducer classes, and the RulesMapper and RulesReducer classes. The logic of these mapper and reducer classes is explained further in Section 4.2. In brief, the first Map-Reduce job finds all frequent items in the dataset, the second job generates all frequent itemsets in the dataset, and the third job generates all association rules on the frequent itemsets. Each mapper class implemented a map() function that mapped each line of each file given as the input to the job to some key-value pair as a mapper output and each reducer class implemented a reduce() function that aggregated all key, value pairs with equal keys to one key and a list of values associated with the key. Some operation was performed on this list of values in the reducer and was then outputted as an output key-value pair. All mapper and reducer classes implemented a setup() method that is called before the map() or reduce() methods are run. This is used to set up preliminary variables in the class.

The Itemset class represents an itemset in the algorithm. Each itemset object consists of a list of items and a support count. The methods in the class are used throughout to examine or add to the internal list of items, to check for equality of itemsets or to update the support count. The Itemset class implements a comparable interface for the purpose of finding the frequent pattern more easily. The compareTo() method compares the support counts of two itemsets.

The AssociationRule class represents a rule in the algorithm. The class consists of an itemset on the left and right of a rule and a confidence level for the rule. The confidence level states the probability of the right itemset appearing in a transaction if the left side is in that transaction and is calculated in the constructor when a rule is made.

The ItemsetUtils class is a static class of useful functions and methods used by the main program. The utility class includes a variety of functions. It is able to read in an Itemset object from its String representation described by the overloaded toString() method in the Itemset class, generate association rules from different combinations of frequent itemsets, return a list of frequent itemsets generated from a base item’s conditional FP-
tree, and even read in and return a list of all frequent items that were found in the first Map-Reduce job.

There are various deprecated methods included in the different classes. These methods are not needed as further design decisions made in later iterations did not need to make use of these functions. This is discussed in Section 4.1.

A ResultsComparator class tests for correctness of the outputs of the implemented algorithm. This class consists of a main method that analyzes the outputs from the implemented algorithm and compares them to the expected outputs. The metrics used to test correctness are discussed in Section 3.3.

4. SYSTEM DEVELOPMENT AND IMPLEMENTATION

The following phases of development followed from the initial requirements gathering discussed in Section 3.

4.1 System Development

4.1.1 Iteration 1: Standard Apriori Prototype

The first iteration of the rule mining module focused on the correctness of the algorithm. To this extent, the standard Apriori algorithm was implemented to mine frequent itemsets from the data and to present as the initial prototype. This prototype was implemented on the MapReduce framework and was able to run as a Hadoop program on a local, single-node Hadoop test environment. Concerns were raised about scalability when the initial prototype was demonstrated.

The standard Apriori algorithm, as mentioned in Section 2, needed to find all combinations of 2 frequent items to find the frequent-2-itemsets, and then find all combinations of 3 frequent items to find all frequent-3-itemsets and so forth. By focusing on this area, an optimization was found where the number of candidate itemsets in each iteration could be reduced. This was done using the Apriori principle where if an itemset is found to be infrequent, all itemsets containing that itemset will also be infrequent [2]. This meant that candidate itemsets would only need to be combinations of frequent itemsets. The optimization in the generation of candidate itemsets, thus, made use of the principle by only generating candidate-k-itemsets that were combinations of the frequent-(k-1)-itemsets with the frequent-1-itemsets per iteration. This led to much fewer candidate itemsets generated per iteration as it only accounts for itemsets that are already found to be frequent rather than generating every \( k \) possible combination of items per \( k^{th} \) iteration.

Regardless of this optimization, the algorithm would still need to iterate through the dataset \( k \) times in order to find all frequent itemsets, where \( k \) would be the number of items in the largest frequent itemset. The generation of frequent itemsets was also found to perform very slowly on datasets larger than 1000 items.

4.1.2 Iteration 2: Sequential FP-Tree Construction

Based on the performance of iteration 1 and the feedback received from the prototype demonstration, it was decided to rather implement the FP-Growth Algorithm for association rule mining to generate frequent itemsets more efficiently.

Due to this design decision, the genCandidateKItemsets() method in the ItemsetUtils class was deprecated as the program would not have to make use of generating combinations of frequent items to find all frequent itemsets. The addSet() method in the Itemset class was also deprecated as a result.

This iteration was also developed on Hadoop’s MapReduce framework. The program consisted of the following phases: the finding of all frequent items, the ordering of transactions based on the frequent pattern, the construction of the FP-tree, the mining of frequent itemsets from the FP-tree, then the generation of association rules from the frequent itemsets.

A problem was encountered where the FP-tree needed to be constructed from the ordered itemsets. The Map-Reduce framework is based on the parallelization of shared-nothing architecture. This meant that Map and Reduce jobs could not share resources when they are run. This was problematic as the FP-tree needed to be constructed with the information of all the ordered transactions in the dataset. It was then decided to implement the construction of the FP-tree as a sequential process after the first two Map-Reduce jobs were run. The conditional pattern bases were then found for each item in the FP-tree and written to an output file. This file was then given to a third Map-Reduce job to mine frequent itemsets from. The generation of association rules was implemented in this iteration as a fourth Map-Reduce job to generate rules on the frequent itemsets.

Tests were then run on the algorithm and it was immediately apparent that the sequential portion of the algorithm was a bottleneck for the performance of the program. This was due to the fact that the algorithm had to essentially run through the dataset twice for the first two Map-Reduce jobs and then run a third time over the dataset sequentially to build the FP-tree. The FP-tree would also take up a big space in memory, especially when dataset sizes were large. The algorithm was run on various datasets to test correctness of itemset and rule generation.

4.1.3 Iteration 3: Map-Reduce Conditional Pattern Bases

After careful consideration, an optimization was found where the sequential portion of the algorithm could be done using the MapReduce framework. This was found from the realization that the construction of the FP-tree in the sequential portion of iteration 2 was only done so that the conditional pattern bases of each item could be used to mine frequent itemsets. The program was then changed to implement the sequential FP-Tree construction as a Map-Reduce job. The specifics are fully explained in Section 4.2.2.
This led to the deprecation of the FPTreeNode class, which was not included in the class diagram in Appendix A. The FPTreeNode class stored information of the structure of the FP-tree generated in the previous iteration but was not needed once it was found that the process could be implemented as a MapReduce job. As a result the reverseSet() method in the Itemset class and the findFreqPatternIndex() method in the ItemsetUtils class were deprecated as well.

The resulting program, thus, consists of three Map-Reduce jobs: the finding of frequent items, the mining of frequent itemsets, and the generation of association rules. This requires the algorithm to only scan through the dataset twice and fully utilizes Hadoop’s capabilities by having the entire program being Map-Reduce jobs. This iteration was then used to perform the evaluation tests that are discussed in Section 5.1.

4.2 FP-Growth Algorithm Implementation

FP-Growth for association rule mining consists of three steps: the finding of all frequent items in the dataset, the generation of frequent itemsets using an item’s conditional FP-tree, and finally, the generation of association rules using the frequent itemsets. The rule mining module of the system implements these three steps of FP-Growth in three Map-Reduce jobs. These jobs utilize the mapper and reducer classes discussed in Section 3.2.

4.2.1 Frequent Items

The first Map-Reduce job finds all frequent items in the dataset. The ItemsMapper class reads one line per map() method from all input files in the directory specified on the HDFS. An offset option is given where the first few items in the line of the input files can be skipped. This offset is used for the IBM dataset as the first two items in each line were the transaction and customer ID’s. In the Twitter data it was also used as each line contained information on the date and time of when the tweet was posted.

After the offset, each item in the transaction was then written as a key-value pair, where the key was the item and the value was the integer constant 1. The ItemsReducer class then aggregates all equal keys to a list of values. In this case, each reduce() method receives a unique item in the dataset and a list of 1’s. The 1’s are summed to form the support count of the item and then written as an output if the support exceeds the specified minimum support threshold.

4.2.2 Generation of Frequent Itemsets

The second Map-Reduce performs many steps of FP-Growth in one job. The OrderedItemsetMapper class uses the original dataset again as an input. It starts off by first initializing the frequentPattern list in the setup() method. The frequentPattern list contains a list of all frequent items that were an output of the first Map-Reduce job. The frequent items are then sorted by support level to form the frequent pattern.

Each map() method works on one line for all files in the input directory, where each line would represent a single transaction. The map() method first constructs the ordered transaction using the order specified in the frequent pattern list initialized in the setup() method. After this is done, the mapper maps each item in the ordered transaction to the prefix of the leading items in the ordered transaction. For example, if the ordered transaction was \{B, H, C, T\}, the map() function would output the following key-value pairs:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>{}</td>
</tr>
<tr>
<td>H</td>
<td>{B}</td>
</tr>
<tr>
<td>C</td>
<td>{B, H}</td>
</tr>
<tr>
<td>T</td>
<td>{B, H, C}</td>
</tr>
</tbody>
</table>

This process results in key-value pairs where all values with equal keys represent the conditional pattern bases of the key. The count of each conditional pattern base can be obtained by counting every occurrence of that conditional pattern base in the list of values per key. This is done by the Reducer.

The FreqItemsetReducer class first starts by constructing the frequent pattern in the setup() method, similar to the OrderedItemsetMapper. The reduce() method then aggregates each item with a list of prefix paths that lead to the item, i.e., the item’s conditional pattern bases. The count of each conditional pattern base is found by counting the occurrence of each unique conditional pattern base in the list of values and is saved in a list. Notice that this provides each reducer with a unique item and its conditional pattern base. Each reducer uses this information to construct the item’s conditional FP-tree from the item’s list of conditional pattern bases and their respective counts and finds all frequent itemsets that contain the base item (key of the reducer’s input key-value pair). These itemsets are then written on one line of frequent itemsets with their counts as outputs.

4.2.3 Generation of Association Rules

The RulesMapper class takes the frequent itemsets generated by the previous Map-Reduce job as an input. The map() method takes all itemsets in a line of the input files and generates combinations of left and right rules. The confidence is calculated using the itemsets used for the generation of the rule and can be looked up in the line of itemsets due to each line being a set of frequent itemsets generated from a common base item. The rule is then mapped with the confidence of the rule as the key-value pair.

The RulesReducer class will then compare each rule with the minimum confidence and write the rule as an output if it exceeds the minimum confidence specified.

5. RESULTS AND FINDINGS

5.1 Evaluation

The rule mining module underwent an evaluation for correctness and scalability. For this purpose, the module was run on two types of datasets. The first dataset is the IBM Quest Synthetic Dataset Generator [15] and the second was the extracted Twitter data provided.
The IBM Synthetic Dataset Generator generates transactional data based on two sets of parameters. The first set of parameters specified how related the items in a transaction were. These include:
- the number of patterns
- the average length of a maximal pattern
- the correlation between patterns
- the average confidence between patterns

The second set of parameters specify the size of the generated dataset. These include:
- the number of transactions to generate
- the average length of a transaction
- the number of different items in the dataset

Once all of these parameters are specified, the Synthetic Dataset Generator creates three files: a dataset file that contains one transaction per line, a configuration file that shows which parameters were used for the dataset, and a pattern file that shows all expected frequent itemsets and the average confidence that rules generated with that itemset should have.

To test correctness of the implemented algorithm, four datasets of varying transactions were generated. The four datasets all shared the following configurations, as presented in Table 4.

**Table 4. Shared configuration parameters of the four datasets.**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of patterns</td>
<td>1000</td>
</tr>
<tr>
<td>average length of maximal pattern</td>
<td>4 (default)</td>
</tr>
<tr>
<td>correlation between patterns</td>
<td>0.25 (default)</td>
</tr>
<tr>
<td>average confidence between patterns</td>
<td>0.75 (default)</td>
</tr>
<tr>
<td>average length of a transaction</td>
<td>10 (default)</td>
</tr>
<tr>
<td>number of different items</td>
<td>1000</td>
</tr>
</tbody>
</table>

Most of the parameters were kept at recommended, default values. The only parameter that was adjusted throughout the four datasets was the number of transactions. From the first to the last the number of transactions was set to 10,000, 100,000, 500,000 and 1,000,000, respectively. The datasets are named using a convention to describe the underlying data. The format lists the three important parameters in order, these being the average transaction length, the average length of a maximal pattern and the number of transactions generated. Thus, the four generated datasets were named T10I4D10K, T10I4D100K, T10I4D500K, T10I4D1000K, respectively.

Correctness was tested by comparing the implemented algorithm’s outputs with the pattern file generated with each dataset. Two aspects were tested: the number of matched itemsets from the implementation’s outputs compared to the itemsets generated in the pattern file, and the average confidence produced from the implementation’s rules compared to the expected average confidence per itemset in the pattern file generated. The metric used in the ResultsComparator class to test confidence was an average confidence deviation offset. This was calculated by first calculating the average confidence of the rules generated from an itemset. All these averages are then compared to the expected confidence levels specified in the pattern file. The absolute value of the difference of these two values is saved in a list for all itemsets in the pattern file and an average is taken from this list of values. This average represents how much the confidence levels generated deviate from the expected confidence values, where a higher average indicates a large deviation from expected values and a low average (ideally zero) would indicate that the rules with generated confidence match the expected confidence values defined by the pattern file of the IBM synthetic dataset. A validation test was also done by running the algorithm on the Twitter data to validate that the algorithm could work on the given data.

Finally, a scalability test was performed on the algorithm. This test was done on Amazon Web Service’s Elastic Map Reduce (EMR) Hadoop cluster. This was chosen since EMR provides dedicated Hadoop-configured cluster that can be run with Hadoop programs on large datasets. This is also the cluster that the Web-based front-end will be interacting with, so the tests emulate real-world conditions. Scalability was tested by calculating the execution time of the implemented algorithm while running on varying dataset sizes, scaling multiplicatively, and on different cluster node configurations. The number of nodes refer to how many machines are working together in a Hadoop configuration. The datasets used were the IBM Synthetic Datasets generated since the specifications of each dataset were made clear from the parameters given to generate them.

### 5.2 Results and Findings

#### 5.2.1 Correctness Evaluation Results

A correctness evaluation was done by testing the T10I410K and T10I4100K datasets with their respective pattern files. The pattern files for the two were identical since both datasets used the same parameters for the pattern generation discussed in Section 3.3. Correctness was tested with two metrics. The first is the number of frequent itemsets that matched with the generated frequent itemsets. The results for this metric are presented in Table 5.

**Table 5. Results of generated itemsets that matched with expected itemsets.**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Expected Itemsets</th>
<th>Generated Itemsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>T10I410K</td>
<td>353</td>
<td>335</td>
</tr>
<tr>
<td>T10I4100K</td>
<td>353</td>
<td>337</td>
</tr>
</tbody>
</table>

The second metric was the test of the average confidence deviation offset, as discussed in Section 3.3. The goal of this metric was to see how much the generated confidence levels deviated from the expected confidence levels for each itemset. A higher deviation offset implies that the confidence in the generated rules deviate greatly from their expected value and a lower deviation offset implies that the confidence levels from generated rules are close to their expected values. These results are presented in Table 6.
and association rules from the datasets.

was chosen to generate a minimum support of 1% configuration.

over three runs of the map and reduce jobs. was the master node and the rest were worker nodes all plotted performance of the algorithm on execution time

S10I4100K

scalability test tested on the m4.large instance type whereas nine, node configurations were tested on the m4.xlarge to numbers of node configurations were tested on the IBM generated datasets.

T10I410K

Dataset
Average Confidence Deviation Offset (%)
T10I410K 13.32
T10I41000K 11.75

The minimum support level was set to 10 and the minimum confidence was set to 50% for both correction metrics in order to find all relevant itemsets and rules.

5.2.2 Scalability Evaluation Results

The scalability evaluation was conducted by running the implemented algorithm on an AWS EMR cluster. The test was done on varying dataset sizes, and various clusters with varying numbers of nodes. Two cluster instance types were used in order to gather data for larger node sizes. Three, five, seven, and nine node configurations were tested on the m4.xlarge instance type, whereas nine, thirteen, and seventeen node configurations were tested on the m4.large instance type. The datasets used for the scalability test were the IBM generated datasets with 10 000, 100 000, 500 000 and 1 000 000 transactions as mentioned in Section 3.3. Figure 3 and Figure 4 present graphs showing the execution time of the algorithm on the varying datasets. The performance of the algorithm on different node configurations are all plotted on the same axes. In each node configuration, one node was the master node and the rest were worker nodes running the map and reduce jobs. All execution times were the average taken over three runs of the module for each dataset for each node configuration.

A minimum support of 1% and a minimum confidence of 50% was chosen to generate a sufficient number of frequent itemsets and association rules from the datasets.

Figure 3. Graph of Execution time vs Increasing Dataset Sizes on Differing Node Configurations on m4.xlarge.

Figure 4. Graph of Execution time vs Increasing Dataset Sizes on Differing Node Configurations on m4.large.

5.2.3 Discussion

It was found for the correctness evaluation tests that the expected itemsets generated from IBM Synthetic Dataset Generator would only generate expected itemsets based on the pattern parameters given and not for the actual dataset that was generated. This had made proving correctness more difficult as it can not be said for all frequent itemsets or rules generated from the implemented algorithm if they are wrong or correct. The best metric for correctness would be to then test for how close the generated outputs matched the expected outputs. Similarly, confidence levels that were generated with expected itemsets could only be tested against. The chosen metric used was an average of the deviations each rule’s confidence had with the expected confidence levels. Due to the fact that the pattern file does not correspond exactly with the dataset generated, one can expect a deviation around 10% to be good for a given dataset.

Regarding the scalability evaluation, it was found during the process of running the tests that many of the Map-Reduce jobs on AWS EMR would prematurely fail due to server issues. This resulted in some of the data points plotted to represent longer execution times than a perfect run of the algorithm. The decision was made to keep all measurements of jobs that succeeded as testing on AWS was also a test in real world conditions where some Hadoop jobs could possibly fail but recover from failure to finish the data mining.

It was also found that the AWS account used to run the scalability tests was limited to the maximum number of nodes that could be requested for a given cluster configuration. This limited the node setups to what is seen in Figure 3, where nine nodes was the maximum that could be assigned for the m4.large instance type. It was decided to further test scalability of larger node sizes on the m4.large instance type, which had permitted up to twenty nodes for a requested cluster. This was used for the measurements taken in Figure 4. It was also found that the m4.large instance type could not handle the computation of larger dataset sizes for a small number of nodes so the nine node configuration was used as a cut off for the two instance types.

Regarding the execution times, it was also found that the reducing phase of the frequent itemsets generation Map-Reduce job would be very slow for the last few reduce methods. This is speculated to
be due to the fact that the keys in the Mapper’s output are sorted internally and then assigned to reducers based on the sorted keys. This can be problematic since the Mapper maps each item to their conditional pattern bases for the reducer to generate frequent itemsets with. This means that items with longer conditional pattern bases require more processing to generate all frequent itemsets for that node. This might cause an uneven load balance for reducer nodes if a node is given many items with larger conditional pattern bases than other nodes.

By examining the execution times presented in Figure 4 compared to the execution times presented if Figure 3, it can be seen more nodes results in a speedup for the performance of the algorithm. However, adding too many nodes would not necessarily lead to a further increase in performance since the 13 node configuration performs better than the 17 node configuration. This could be attributed to the fact that there is overhead time taken to assign workloads when more nodes are involved in the processing of a task which leads to longer execution times.

There is also a tradeoff to be observed when choosing the performance capabilities of each node. The m4.xlarge instance type offers more powerful nodes than the m4.large instance type, but an m4.large cluster was observed to out-perform an m4.xlarge cluster if given more nodes. This is seen in the computation times of Figure 4 when compared to the computation times of Figure 3.

The results found from the scalability tests show the algorithm’s performance using a minimum support level of 1%. This implies that the algorithm would generate and mine frequent itemsets from items that are contained in at least 1% of the transactions. Due to the nature of the algorithm, this means that one can expect computation times shorter than the times found in the results if a higher minimum support level is used.

The algorithm is seen to scale exponentially with the dataset but datasets with transactions in the millions can still be processed on the order of minutes by observing the trajectory of the graphs. Clusters with higher node configurations can be seen to scale slower, which indicates that higher node counts increase performance with larger datasets. However, it is seen that there is a tradeoff in the number of nodes that should be assigned as there is overhead time associated with managing more nodes in a cluster.

6. CONCLUSIONS

The implemented rule mining module underwent many iterations where the final iteration took full advantage of Hadoop’s environment by using fully implemented map and reduce methods that could be split for all nodes to process. It was also immediately apparent that the FP-Growth algorithm performed much better than the standard Apriori algorithm in terms of execution time when generating frequent itemsets. This was observed when comparing the execution time to generate frequent itemsets at the end of Iteration 1, discussed in Section 4.1, with the execution times of the final version of the mining module presented in Figure 3 and 4. The execution times for Apriori were found to take on the order of minutes for a dataset of 1000 transactions as presented in the evaluation demonstration for Iteration 1, where the worst case for the final version was 900 seconds, or 15 minutes for a dataset of 1 000 000 transactions.

The correctness results have sufficiently matched most generated itemsets with the expected itemsets, as shown in Table 5. The deviation of confidence levels is also observed to shrink as the dataset increases which indicates that rules generated can be more accurate with bigger data sizes. This is shown in Table 6 where confidence deviation averages are shown for a growing dataset.

Regarding the scalability of the algorithm, an improvement can be seen when increasing the number of nodes. Particularly, the 13 node configuration is observed to have the shortest computation times for larger datasets being roughly 400 second less than the worst case for 1 000 000 transactions (7 node configuration in Figure 3) and can be visually seen as a major improvement from the 17 node configuration in Figure 4.

Since the developed rule mining module makes full use of Hadoop’s Map-Reduce framework, the computation time is directly affected by the specifications of the cluster it is run on. If more powerful nodes could be assigned and more resources could be allocated to a cluster, it is expected that the module would see an increase in performance and would be able to handle dataset sizes larger than the ones used for testing.

REFERENCES


DOI: http://doi.org/10.1145/2492517.2500301


DOI: https://doi.org/10.1109/69.553164