A Fast Algorithm for Data Mining

CS 297 Report

Aarathi Raghu

Advisor: Dr.Chris Pollett

December 2005

A Fast Algorithm For Data Mining

Abstract

This report describes the data mining algorithms implemented and lessons learned during the course of my CS 297. Data Mining is a growing field and a plethora of algorithms have been proposed. The Apriori algorithm is a commonly used algorithm in data mining. We studied modifications of the Apriori algorithm and a lattice based algorithm in this semester. Implementation of these algorithms formed three of the four deliverables for this semester. So far, we have successfully completed the three deliverables- the bitmap based apriori and the disk-based improvement of the previous implementation. We are in the process of completing the fourth deliverables. The last deliverable will be the foundation for work in CS298.

Introduction:

A problem with very large databases (databases with terabytes of data) is data mining. Data mining is the ability to derive from the database meaningful data. Data mining tools help in knowledge discovery and help in assimilating trends, or deriving association rules from the databases. The main advantage with being able to mine this kind of information from datasets is that businesses can gauge accurately how their businesses are running and assess how they can be improved, based on these rules. Generating these rules, however, is time consuming. Some of these algorithms, unfortunately, run in exponential time. This semester, we attempted to implement several algorithms, each of which will be described in detail in following sections. We also intend to test different kinds of datasets to look for improvements in speed.

A common algorithm in Data Mining is the Apriori algorithm. This algorithm was pioneered by Agarwal and Srikant, and has hence been modified by various groups. The algorithm is based on generating frequent itemsets, and a byproduct of this is also mining deriving the association rules.

Lin and Louie have made contributions to data mining by way of granular computing. Their algorithm is based on generating association rules based on granular computing. Deliverable 1 is based on this algorithm and will be described in the next section.

Lin, Hu, and Louie proposed another algorithm, wherein they considerable reduced the search space by employing a breadth first search to construct the basis of a lattice. This basis is used to form an

association rule. Implementing this algorithm turned out to be deliverable 3, which I will describe in an ensuing section.

The above algorithms are in exponential time. However, there may be certain distributions of data for which these algorithms may be faster. We are yet to try out the lattice-based algorithm with a Zipf distribution to look for improvements in the running time. Based on this, we propose to improve the algorithm as part of the CS298 project.

In the following sections, we will describe the motivation, goal, and results of each of the deliverables, and basis for work in CS298.

Deliverable 1:

Motivation:

The motivation for this deliverable was to get an understanding for a commonly used algorithm in data mining, namely the Apriori algorithm [Agarwal]. Before implementing this algorithm, we read chapters that described the basics of datamining [Ramakrishnan],[Molina]. Lin and Louie's algorithm [Lin2002] using bitmaps was faster than the original Apriori algorithm and hence we decided on their algorithm as the basis for our work**.** Since, we had decided to go with the BitMap approach, we wanted to see if we can do better than Java's BitSet implementation for basic bit manipulations. For all of the common operations such as "and", "or", "cardinality", "set", and "clear", we timed the native implementation and our implementation and for the most part, we had better timing than the native implementation.

Goal:

The purpose of this deliverable was to implement the Apriori algorithm based on bitmaps (granular computing). We also wanted to test the scalability of this algorithm and get different modules in place.

Implementation details and results:

The program can roughly be divided into three modules : one module ReadFile.java that reads in a file into memory, a driver program Apriori.java that makes a call to ReadFile.java and algorithm.java, and the third module algorithm.java that contains the core of this implementation by constructing the granular model after reading in the table from the previous module. The algorithm computes the bitmap for each distinct attribute value. This is done in the following way. Every column (tuple) will be shrunk into bit vectors corresponding to each unique value in the column. Only those bit vectors that are above the threshold value are retained and this reduced data set is the 1-large itemset. A large itemset is defined as an itemset whose bitmap representation has cardinality greater than or equal to the threshold, that is specified in the program. Since we use bitmaps, computing the cardinality involves counting the number of set bits.

The next step is to compute the 2-large itemsets. This is done by computing the intersection between any two unique 1-large itemsets. Using the bitmap representation speeds up the process. Any two one-large itemsets are intersected and those values above the threshold constitute the 2-large itemsets.

In general, the computation of any n-large itemset is done recursively, where the n-large itemset is computed by intersecting an (n-1) large itemset with a 1-large itemset.

This algorithm had the basic structure in place. However, we could still tweak a lot of parameters to make this a better implementation. One of them was to stop fetching the entire file into memory. For a very large database, which we are yet to test with, this algorithm will not scale. So, one of the changes we inevitably had to make was to make this algorithm disk-based, as we will do with deliverable 2. Also, we thought we could do better than Java's BitSet code, which we had initially used, in terms of performance. We implemented our own ByteClass class which we later used.

The test data set for this algorithm is a set of randomly generated integer values. It consists of 5 columns and 20 rows. Here is a sample output we get while running this algorithm. The values on the left are the columns to which the bitmaps belong to. We begin with generating the 1-large itemsets, followed by 2-large and so on.

```
Generating one-large item sets
[0]: 00011110000000000000
[0]: 00000001111000000000
[0]: 11100000000111000000
[0]: 00000000000000111111
[1]: 00000000000111000000
[1]: 11111110000000000000
[1]: 00000000000000111111
[1]: 00000001111000000000
[2]: 00000001111000000000
[2]: 00000000000111111111
.....
[4, 1]: 11100000000000000000
[4, 1]: 00000000101000000000
[4, 3]: 00011110010000000000
[4, 3]: 11100000000000000000
[4, 3]: 00000000000100011111
[1, 0]: 00000001111000000000
[1, 0]: 00011110000000000000
[1, 0]: 11100000000000000000
[1, 0]: 00000000000111000000
[1, 0]: 00000000000000111111
[2, 0]: 00000000000111000000
```

[2, 0]: 00011110000000000000
[2, 0]: 00000001111000000000
[2, 0]: 00000000000000111111
[2, 0]: 11100000000000000000
[3, 0]: 11100000000000000000
[3, 0]: 00011110000000000000
[3, 0]: 00000000000111000000
[3, 0]: 00000000000000011111
[4, 0]: 11100000000010000000
[4, 0]: 00011110000000000000
[4, 0]: 00000000000000111111
[4, 0]: 00000000101000000000
Done with 2 large item set: 41
[2, 3, 0]: 00000000000000011111
[2, 3, 0]: 11100000000000000000
[2, 3, 0]: 00011110000000000000
[2, 3, 0]: 00000000000111000000
[2, 4, 0]: 11100000000000000000
[2, 4, 0]: 00000000000000111111
[2, 4, 0]: 00011110000000000000
[2, 4, 0]: 00000000101000000000
[4, 1, 3]: 00000000000000011111
[4, 1, 3]: 11100000000000000000
[4, 1, 3]: 00011110000000000000
[2, 1, 0]: 00000000000111000000
[2, 1, 0]: 00000001111000000000
[2, 1, 0]: 11100000000000000000
[2, 1, 0]: 00011110000000000000
[2, 1, 0]: 00000000000000111111
[1, 3, 0]: 11100000000000000000
[1, 3, 0]: 00000000000111000000
[1, 3, 0]: 00011110000000000000
[1, 3, 0]: 00000000000000011111
[4, 1, 0]: 00000000000000111111
[4, 1, 0]: 00000000101000000000
[4, 1, 0]: 11100000000000000000
[4, 1, 0]: 00011110000000000000
[4, 3, 0]: 00000000000000011111
[4, 3, 0]: 00011110000000000000
[4, 3, 0]: 11100000000000000000
[2, 1, 3]: 00011110000000000000
[2, 1, 3]: 00000000000111000000
[2, 1, 3]: 11100000000000000000
[2, 1, 3]: 00000000000000011111
[2, 4, 1]: 00000000101000000000
[2, 4, 1]: 00011110000000000000
[2, 4, 1]: 11100000000000000000
[2, 4, 1]: 00000000000000111111
[2, 4, 3]: 00000000000100011111
[2, 4, 3]: 00011110000000000000
[2, 4, 3]: 11100000000000000000
Done with 3 large item set: 38
[4, 1, 3, 0]: 00000000000000011111
[4, 1, 3, 0]: 00011110000000000000

```
[4, 1, 3, 0]: 1110000000000000000
[2, 4, 1, 3]: 0000000000000011111
[2, 4, 1, 3]: 0001111000000000000
[2, 4, 1, 3]: 1110000000000000000
[2, 1, 3, 0]: 0001111000000000000
[2, 1, 3, 0]: 0000000000111000000
[2, 1, 3, 0]: 1110000000000000000
[2, 1, 3, 0]: 0000000000000011111
[2, 4, 1, 0]: 1110000000000000000
[2, 4, 1, 0]: 0000000000000111111
[2, 4, 1, 0]: 0001111000000000000
[2, 4, 1, 0]: 0000000101000000000
[2, 4, 3, 0]: 0000000000000011111
[2, 4, 3, 0]: 1110000000000000000
[2, 4, 3, 0]: 0001111000000000000
```

Deliverable 2:

Motivation:

The motivation behind this deliverable was to improve on the scalability and performance of deliverable 1. Although, we had implemented the bitmap-based Apriori algorithm, we wanted an algorithm that could generate an n-large itemset, where n is large. Additionally, we wanted the algorithm to generate n-large itemsets for larger datasets. The basis for this work was adapted from Lin and Louie's paper [Lin2002].

Goal:

The goal for this deliverable was to tweak the previous implementation in such a way that we simulate disk reads by reading in a block of data at a time.

Implementation and results:

This deliverable had the following programs:

DiskReader.java:

This program simulates disk-reads by reading in data from a file into memory in 4K chunks. The 4K chunk of data in memory is used to build the Granular model directly. Once this is built, the next 4K chunk is

fetched from disk. This ensures that we use memory judiciously, especially when we are dealing with large datasets.

ItemSetInfo.java:

This program implements the data structure for holding the bitmaps corresponding to each unique value in a column.

Algorithm.java:

In this program, we set a variable maxValsPerColumn that keeps track of the maximum number of (n – 1) large itemsets before we move on to n – large itemsets. Limiting the number of (n-1) large itemsets is beneficial because we can index into an array to generate the n large array by intersecting the (n-1) large and 1-large itemsets. This array is a two dimensional array in which the first dimension keeps track of which large itemset we are building and the second dimension keeps track of the values obtained by intersecting 2 columns. This dimension has a maximum index which limits how many values we generate. Though limiting the number of values hinders completeness of results, it ensures better scalability by reducing memory usage.

We tried this algorithm out with a dataset as large as 200 columns * 200 rows and we could generate as many as 15-large itemsets. Here is a sample output for a smaller dataset. The values on the left show the attribute value corresponding to the bitmap. The maxValsPerColumn was set to 100 for this run.

```
2 :00000001111000000000
8 :00011110000000000000
9 :11100000000111000000
5 :00000000000000111111
2 :00000000000000111111
3 :11111110000000000000
5 :00000000000111000000
0 :00000001111000000000
15 :11100000000000000000
8 :00011110000000000000
```

```
12 :00000000000111111111
0 :00000001111000000000
2 :00011110010000000000
4 :00000000000111011111
5 :11100000000000100000
8 :00000000000100111111
9 :11100000000010000000
6 :00011110010001000000
7 :00000000101000000000
Done...19
Done with one large item set...
Generating 2 large item set
2 0 :00000001111000000000
8 3 :00011110000000000000
9 3 :11100000000000000000
9 5 :00000000000111000000
5 2 :00000000000000111111
2 0 :00000001111000000000
8 8 :00011110000000000000
9 15 :11100000000000000000
9 12 :00000000000111000000
5 12 :00000000000000111111
8 2 :00011110000000000000
9 4 :00000000000111000000
9 5 :11100000000000000000
5 4 :00000000000000011111
2 7 :00000000101000000000
8 6 :00011110000000000000
9 9 :11100000000010000000
5 8 :00000000000000111111
2 12 :00000000000000111111
3 15 :11100000000000000000
3 8 :00011110000000000000
5 12 :00000000000111000000
0 0 :00000001111000000000
2 4 :00000000000000011111
3 2 :00011110000000000000
3 5 :11100000000000000000
5 4 :00000000000111000000
2 8 :00000000000000111111
3 9 :11100000000000000000
3 6 :00011110000000000000
0 7 :00000000101000000000
15 5 :11100000000000000000
8 2 :00011110000000000000
12 4 :00000000000111011111
15 9 :11100000000000000000
8 6 :00011110000000000000
12 8 :00000000000100111111
0 7 :00000000101000000000
2 6 :00011110010000000000
4 8 :00000000000100011111
5 9 :11100000000000000000
Done...41
Generating 3 large item set
```

```
2 0 0 :00000001111000000000
8 3 8 :00011110000000000000
9 3 15 :11100000000000000000
9 5 12 :00000000000111000000
5 2 12 :00000000000000111111
8 3 2 :00011110000000000000
9 3 5 :11100000000000000000
9 5 4 :00000000000111000000
5 2 4 :00000000000000011111
2 0 7 :00000000101000000000
8 3 6 :00011110000000000000
9 3 9 :11100000000000000000
5 2 8 :00000000000000111111
8 8 2 :00011110000000000000
9 15 5 :11100000000000000000
9 12 4 :00000000000111000000
5 12 4 :00000000000000011111
2 0 7 :00000000101000000000
8 8 6 :00011110000000000000
9 15 9 :11100000000000000000
5 12 8 :00000000000000111111
8 2 6 :00011110000000000000
9 5 9 :11100000000000000000
5 4 8 :00000000000000011111
Done...24
Generating 4 large item set
8 3 8 2 :00011110000000000000
9 3 15 5 :11100000000000000000
9 5 12 4 :00000000000111000000
5 2 12 4 :00000000000000011111
2 0 0 7 :00000000101000000000
8 3 8 6 :00011110000000000000
9 3 15 9 :11100000000000000000
5 2 12 8 :00000000000000111111
8 3 2 8 :00011110000000000000
9 3 5 15 :11100000000000000000
9 5 4 12 :00000000000111000000
5 2 4 12 :00000000000000011111
8 3 2 6 :00011110000000000000
9 3 5 9 :11100000000000000000
5 2 4 8 :00000000000000011111
8 3 6 2 :00011110000000000000
9 3 9 5 :11100000000000000000
5 2 8 4 :00000000000000011111
8 8 2 6 :00011110000000000000
9 15 5 9 :11100000000000000000
5 12 4 8 :00000000000000011111
Done...21
Generating 5 large item set
8 3 8 2 6 :00011110000000000000
9 3 15 5 9 :11100000000000000000
5 2 12 4 8 :00000000000000011111
8 3 8 6 2 :00011110000000000000
9 3 15 9 5 :11100000000000000000
5 2 12 8 4 :00000000000000011111
```

```
8 3 2 8 6 :00011110000000000000
9 3 5 15 9 :11100000000000000000
5 2 4 12 8 :00000000000000011111
Done...9
```

Deliverable 3:

Motivation:

While the above two deliverables are data mining algorithms, they are more relevant while considering smaller datasets. The reason behind this is that these algorithms need to generate frequent itemsets one at a time, and this could be resource-intensive as datasets get larger. Lin, Hu, and Louie, described an attribute-value lattice [Lin2003] for mining association rules, in which the search-space for finding frequent itemsets is reduced considerably. This factor single-handedly makes this a more feasible algorithm for data-mining.

Goals:

Our goal was to implement this algorithm to use this as a basis for future work. We intend to test this algorithm out with different datasets to check how the algorithm will respond to different kinds of datasets. We are yet to check the running time of this algorithm for different datasets.

Implementation and Results:

The attribute value lattice consists of a set of nodes, where each node is set at a certain level and has a keyset and a bitmap which tracks the rows in which the keyset appears. For graph construction, each node also tracks its parents.

For generating the attribute value lattice, we used the DiskReader class which was developed for deliverable 2. The nodes in the lattice are built by looking at the bitmaps and considering only those that exceed the

minimum support value specified in the algorithm. We construct the attribute value lattice based on Lin, Hu and Louie's paper [Lin2003]. The next step is to find the closed frequent itemsets, which form the association rules we are looking for. This is done by looking at the level 1 nodes (which form the greatest lower bounds) and making sure combinations of these are over the minimum support. If they are, then the parents are automatically considered to be part of the closed frequent itemsets. If they are not, then the sibling's parent in combination with the node itself is tested to check if the combination exceeds the minimum support. The algorithm runs until all the level 1 nodes and parents (of sibling nodes) are exhausted. The reduced number of nodes we consider makes this algorithm faster than the previous two.

Future Work:

We will test this algorithm with different kinds of datasets to suggest improvements to this algorithm. Using data that follows the Zipf distribution is one such test. We also will test this algorithm in different ways to check if this will run in polynomial time with the necessary improvements made and for certain kinds of data. This will be done in CS298.

References:

[Ramakrishnan] R.Ramakrishnan and J.Gehkre. Fundamentals of Database Systems.McGraw-Hill, 2002

[Molina] H.Garcia-Molina, J.Ullman, and J.Widom. Database System Implementation.Prentice-Hall, 2000

[Agarwal1994] R.Agarwal, and R. Srikant. Fast Algorithms for Mining Association Rules. Proc. Intl. Conf. on Very Large Databases. pp1522-1534.

[Lin2003] T.Y.Lin, X.T.Hu, and E.Louie. Using Attribute Value Lattice to Find Frequent Itemsets. Data Mining and Knowledge Discovery: Theory,Tools and Technology. 2003.pp 28-36.

[Lin2002] T.Y.Lin, and Eric Louie. Finding Asscoiation Rules by Granular Computing: Fast Algorithm for finding association rules. Data Mining, Rough Sets and Granular Computing. 2002.pp 23-42.