RANDOMIZED DECISION TREES FOR DATA MINING

By

VIDYAMANI PARKHE

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2000

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to Dr. Raj for his support and encouragement, throughout my graduate program at the University of Florida. It was due to his motivation and advice that the challenges of initially a transfer to CISE and then a master's program, were never a hurdle. I am indebted to Professors Sumi Helal and Manuel Bermudez for their undepletable attention and their great help, since the very first day at UF. I thank Professors Sartaj Sahni and Joachim Hammer for being on my committee and their suggestions and comments.

There are a few people, to whom I am grateful for multiple reasons. Firstly, my family back home in India—without their support and trust, nothing would have been possible. Next, my closest ever friends—Amit, Latha, Sangi, Prashant, Mahesh, Prateek, Subha and Hari amongst others, for being my family here. Without their support, encouragement and wrath the journey would have been an impossible one. Special thanks go to John Bowers, Nisi and Victoria for being there, always!

I cannot stop short of thanking Leo's, Chilis and Taco Bell for their extended hours and exquisite food, that has now become an integral part of our lives.

TABLE OF CONTENTS

CHAPTERS

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

RANDOMIZED DECISION TREES FOR DATA MINING

By

Vidyamani Parkhe

December 2000

Chairman: Sanguthevar Rajasekaran
Major Department: Computer and Information Science and Engineering

Classification data mining is used widely in the area of retail analysis, disease diagnosis and scam detections. Of late, the application of classification data mining to the area of web development, web applications and analysis is being exercised. The major challenges to this new facet of classification are the enormous amount of data, data inconsistencies, pressure for time and accuracy of prediction. The contemporary algorithms for classification, that majorly use decision diagrams, are less useful in such a scenario. The major impediment is the large amount of static time required in building a model (decision diagram) for accurate prediction or decision making at run-time and the lack of an efficient incremental algorithm. Randomized and sampling techniques researched for the problem have been less accurate. The present work discusses deterministic and randomized algorithms for classification data mining that are easily parallelizable and have better performance. The algorithms suggest novel methods, like multiple levels of intelligent sampling and partitioning, to collect record distributions in a database, for faster evaluation of gini indexes. An incremental algorithm, to absorb newly available data-sets, is also discussed. A combination of these characteristics, alongwith very high accuracy in decision-making, makes these algorithms adept for data mining and more specifically web mining.

Key words: Classification, data mining, randomized algorithms, decision diagrams, incremental algorithms, gini index and web mining.

CHAPTER 1
INTRODUCTION

With the current trend in the industries to learn from one's mistakes and those of others, setups from small retailers to large corporations are looking towards the "Knowledge Discovery" paradigm to get a more comprehensive overview of their own data. This has been further made possible due to the increased performances in the data warehousing and data mining algorithms, falling costs of storage units and incresed processing speeds. The sections that follow illustrate techniques used to store and mine data, in order to derive information therefrom, that had never been available or thought of heretofore.

1.1 Data Warehousing

With the excessively large customer transactions happening every second, in large super markets, internet sites, bank, insurance or phone companies, there is a need to come up with alternative methods to store the historical data, so that there is no loss of information. Also there could be a need, at a later date, to draw out the *hidden knowledge* from the data without a tangible loss of information. The study of data warehousing comprises everything from the machine architecture and the data store, that could be best compatible to store the specific form of data, to the algorithms used to handle and store data.

A data warehouse can be thought of as a collection of different sources of data, put together, that have been cleaned and checked for inconsistencies prior to merging. These data sources could be from different locations of a chain of super markets, like Wal-Mart, or data recorded at the same location over time. The need to remove the inconsistencies in the data is dire, otherwise the information, drawn out after mining the entire data, would be incorrect. There are various algorithms used for the purpose of data source cleaning, data merging and storing it in a format, from where the applications[1] using the data can access it, in the most efficient manner. Such algorithms have been documented in literature [1, 2]. Widom [2] also quotes a few issues in data warehousing designs that significantly affect the application using the data.

1.2 Data Mining

Data mining can be viewed as a application that resides over a data warehouse and uses the data to search for certain **unknown** patterns. The patterns could be in the form of rules or clusters or some classification, as described in the following sub-sections. Data mining is different from OLAP,[2] in that OLAP uses the query techniques to confirm the results known in the past or by heursitics.

In opposition, data mining is indeed a search for the unknown, wherein the entire data set is used to draw out some information about it, at large, rather than the specifics of the data in itself.

In the following sub-sections we will have a closer look at the various techniques used for data mining .

1.2.1 Assocation Rules

This form of data mining is used to relate two or more quantities together, that otherwise would apparently not have co-existed. An example would be 60% of the people buying *beer* also buy *diapers* and 2% of all the transactions happening contain both beer and diapers. This can be stated in the form a rule, $Beer \Rightarrow$

[1] There could be varoius applications that could make use of the data store, like OLAP, data visualization, data mining or even transactional ones for day-to-day transactions.

[2] On-line Analytical Processing.

Diapers. This rule is said to have a 60% *confidence* and 2% *support*. The task of Association Rule Mining is to come up with a set of rules that satisfy a minimum confidence and support level. One of the leading algorithms used for Association Rule Mining is the Apriori Algorithm [3, 4, 5].

1.2.2 Clustering

The principle used in clustering is to group together (or cluster) the data points that have a common characteristic. The whole idea is to partition the entire data set into categories, depending upon some feature(s), such that the items in one group or cluster are *more* similar to those in the same cluster as compared to the ones in other clusters. Clustering is used in various facets of knowledge discovery and learning, like machine learning, pattern recognition, optimization ...etc. A classic algorithm for clustering starts off with designating k data points that act as centroids for the k clusters, and proceeds with evaluating the nearest centroid for every data-point (to be clustered) and then re-evaluating the mean (centroid) of the data-points in that cluster.

1.2.3 Sequential Patterns

Initially proposed by Agarwal and Srikant [6], the technique uses time (in most cases) to detect a similarity (pattern) in the occurrence of events. This can be used in various applications, like detecting patterns in which books are being read by a set of library users, or detecting a chain-referral scam of medicine practioners, or even more critical ones like disease diagnosis [3].

1.2.4 Classification

"One's ability to make the correct set of decisions while solving a certain problem and doing so in the specific allotted time is the key factor in one's success, time and again. This is true for any sort of problems—may it be from a research perspective or a political one—only the variables and constants change."

–*Anonymous*

Classification data mining is associated with those aspects of knowledge discovery, wherein there is a need to categorize data points, associating them with a certain classification or category, based upon the classification of a few known data points. Here, the objective is to traverse the *training*[3] data set and coming up with a model that could be used for future classification of a *test*[4] data set. The technique of classification has been used since a very long time for the purpose of machine learning [7], optimizations using neural networks, decision trees . . . etc.

A decision tree or a decision diragram comprises a root node, that one uses to make his *first* decision, while classifying a *test* record. A non-leaf node in the decision diagrams represents the data reprsented by its sub-trees, while a leaf node represents data belonging to **one** class and satisfies the conditions of all its ancestor nodes. The decision is **binary**, in most cases, in that it could be either *true* or *false* which takes one to one of the sub-trees of the root, for which the same procedure could be recursively applied, until one reaches the leaf node, that determines the classification for the record under consideration. The non-leaf nodes are decision-making nodes (mostly binary) and hold a condition like $Age \leq 25$, which would lead on to two sub-trees, the data in which always satisfies the condition laid forth by the common parent node, i.e. $Age \leq 25$.

1.3 Goal

Most of the contemporary algorithms for growing decision trees discuss cost effective methods, by which the size (height and spread) of the tree is optimized

[3] A data set for which a classification is known.

[4] A data set for which the classification is not known and has to be determined.

and so is the dynamic time required for decision making. The algorithms output the most compact form of a tree for a given training set, but are cost-inefficient at the tree-growing stage. The principle used in most of the algorithms is to come up with the best split attribute for a given data-set, about which the entire data-set could be categorized into two sections (for binary decision diagrams), such that most of the records of a type belong to one of the sub-trees. Now, for a given data-set, it is very time consuming to come up with the very **best** split attribute at every stage in the tree. Gini values are used (in most cases) to determine the best split at a certain stage. The detailed explanation for gini values and their calculation would be a subject for the following chapters.

The approach, mentioned above, is acceptable and often used in situations where the decision diagrams are made statically and then used for the purpose of decision making at run-time. Also, if there is no need to update the classification for a long time, an approach that gives the most succinct tree is required, as then, the time required for making decisions at runtime would be largely reduced. But, in situations where it is necessary to update the decision diagram very frequently, it might be required to come up with an approach that does so, in a very short interval of time. As discussed before, the most time-consuming task, in the tree building stage, is to determine which is the best *split attribute-value pair.* If one spends time in deciding over the very best split at every stage, undoubtedly the most concise and compact form of the tree would be obtained, but this could be heavily time consuming. Conversely, if the very best split is not determined the trees tend to be wider and longer (increasing the time required to make decisions at run-time, while using the tree).

If the decision diagrams are to be used for the purpose of making the most critical decisions, that have a very high risk level, one might be more inclined to use an algorithm that, although takes a lot of static creation time, outputs the most concise and compact embodiment of the life-critical data set, a query on which would not take long to execute. On the other hand, an application that has no critical hazard might be greatly helped by an inherently incremental algorithm, which would help in assimilating and consuming the most recent data within the decision diagram. Thus, the trade-horses of creation and usage time are determined depending upon the application in mind and ultimate usage.

Web-applications, in most cases, are like the latter ones described above. An example would be click-stream analysis, wherein the objective is to observe a pattern from the web-clicks of various users to a set of web-pages. The problem can be states as follows:

Imagine yourself to be an owner of a web-based commerical store that sells books. There is a group of **loyal** customers that can be identified using their login names and passwords. Every click made by every person, till date, has been recorded. This comprises a range of customers that merely surf through the web-pages under your company's domain and buy nothing and others that are avid buyers. Would it not be a interesting piece of knowledge to know who is buying exactly what, and more specifically if there is a pattern of the types of books being bought by various customers over a specific range of time! It might prove to be commerically advantageous to be able to predict the buying pattern of a set of customers (or potential customers) depending upon the buying-patterns of other customers. But, with the amount of clicks being made on the web-pages and the increasing number of transactions happening every second, it might be impossible, at run-time, to search for and identify the parallelism between a set of clicks of one particular customer and another, in the past. An effective data-structure like a decision diagram would certainly come handy in such cases, where it is most important to interest a customer more in what he would have otherwise, anyway been interested in, and making business. In such a case, the problem is mostly one sided, here the presence of a *decison-maker* or a *next-click-predictor* is not of primary importance, but having one such (a good one) would definitely help in growth of business.

Also, as in case of the click-stream example above, a algorithm that is incremental, in that it can incorporate fresh data into the decision-making data structure, would be of significant use, rather than having a static decision-maker that reflects the choice and trends in the market from a earlier era. As in the above case, it could prove advantageous to be able to make decisions based on some clicks, results or transactions happening just the previous second![5]

The current work concentrates on the issues mentioned above, using techniques like randomized algorithms and sampling to achieve speedups, without the loss of accuracy. An attempt is also made, at making the algorithms incremental, so that any additions to the data sets could be reflected in the decision-maker (also referred to as the learner). In the chapters that follow, the algorithms and the implementation details are mentioned, giving details of the data structures used for the purpose.

[5] Though, it might be difficult and highly cost-inefficient to try and accomodate data from a transaction that occured just a few minutes back.

CHAPTER 2
RELATED WORK IN THE AREA OF CLASSIFICATION DATA MINING

Decision diagrams and other classifiers like genetic algorithms, Bayesian and neural networks have been used for a very long time for the purpose of simple classification and decision support. Anahory and Murray [1] give detailed analysis as to how one could use tools like decision diagrams for data mining which can be very effective in the case of decision support over a data warehouse. Since the evolution of the decision diagrams, a lot of algorithms, varying in time complexity and the kind of data to be classified, have been devised for the purpose of classification. Some of the famous algorithms developed include ID3 [8], C45 [7], SLIQ [9], SPRINT [10], CLOUDS [11], and others. All these algorithms and other previous work in the area of classification data mining have sought the best possible way—given a data-set or a database of records—to provide the classification with the most concise representation or data-structure. Some of the common forms of representation used for the purpose of classification data mining are neural networks, decision diagrams . . . etc. Another issue of primary importance is the time required to pack the given data-set in the selected format of representation, in the minimum possible time frame.

In all the algorithms to build decision diagrams, mentioned above, a common premise and one of the most important objective is that the tree building algorithm should be precise. The tree should be an exact representation of the given *test* data-set. But, in the race to come up with a perfect tree, a lot of time is spent in building the tree in the first place. These algorithms are cost effective and suggest techniques like parallel and simultaneous execution for a faster growth in the tree as in the work by Shafer et al. [10] but a certain amount of time has to be spent at every node in the tree for the determination of the best split—which cannot be compromised.

In the sections that follow, a brief description of a few predominantly used algorithms for classification data mining is given. These include SLIQ, SPRINT and CLOUDS. All of the above use the *Gini Index* for estimation of the best split attribute and value.

2.1 Gini Calculation

One of the most important aspects of building a decision tree is to determine the best *split attribute-value* pair for a certain data-set. Thus, at every stage in the formation of the decision tree, the attribute-value pair that gives rise to the best split of the current data-set has to be determined. The importance of estimating the best split is that otherwise, the resultant tree could be longer and wider. In the worst case, the resultant tree could be a skewed one, with only one non-leaf child per node and a smaller number of records being classified at each step, as depicted in the figures below. Table 2-1 shows a sample data-set that needs to be classified, and Figures 2-1 and 2-2 show a concise and a skewed representation of the same data-set.

The *Gini Index* for a data-set is defined as follows: Consider a data-set S consisting of n records, each belonging to one of the c classes. The *gini* index for S is defined as

$$gini(S) = 1 - \sum_{j=1}^{c} p_j^2$$

where p_j is the relative frequency of class j in S.

To utilize this for the estimation of the best split, consider that a *split*

Table 2.1: Sample dataset for gini calculation

A	B	C	Class
5	2	1	1
2	1	3	2
2	2	1	1
2	1	1	3
1	1	2	2
3	2	4	2
5	4	3	1
8	4	2	1
2	2	1	1
2	3	3	3
2	4	5	2

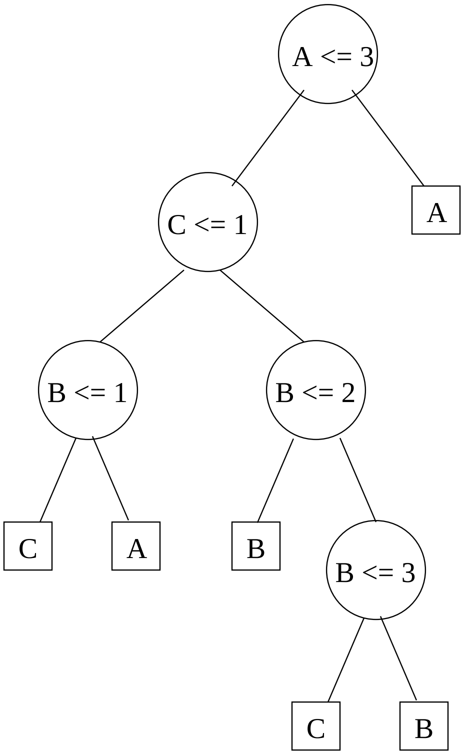


Figure 2.1: Compact Tree

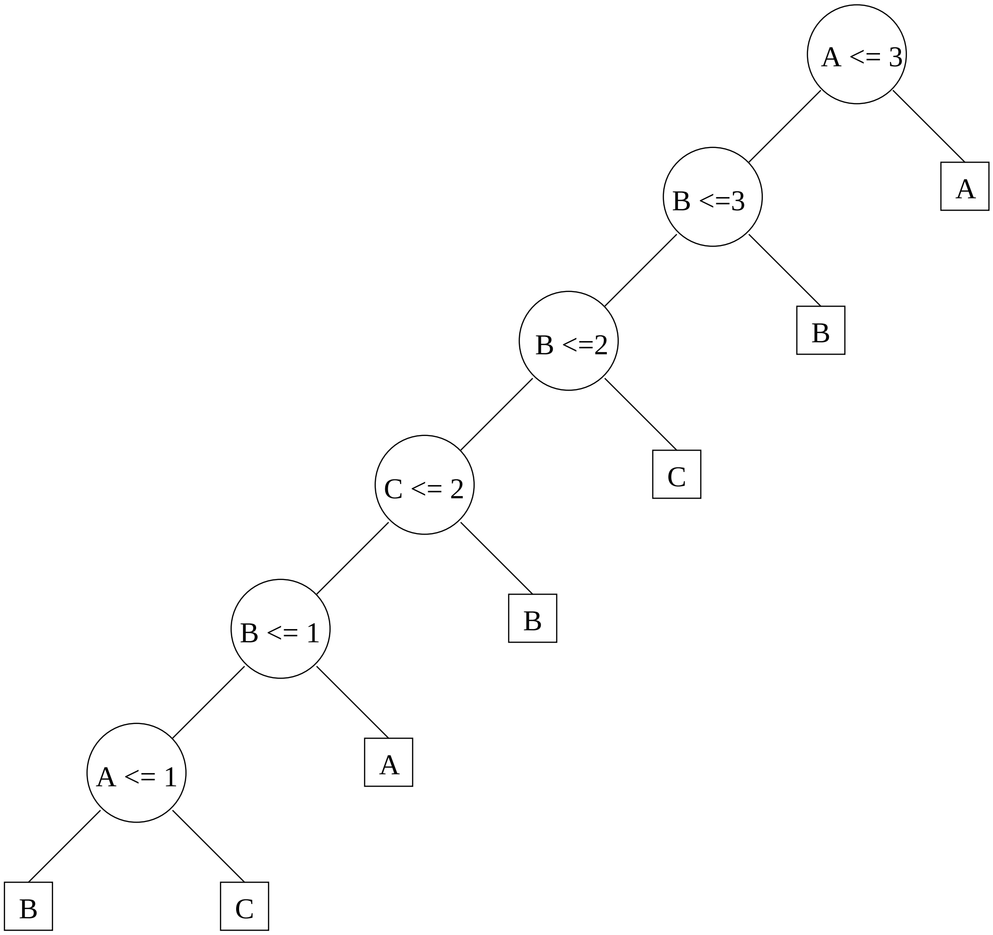


Figure 2.2: Skewed Tree

partitions S into S_1 and S_2. The gini value of the split can be estimated using

$$gini_{split} = \frac{n_1}{n}\, gini(S_1) + \frac{n_2}{n}\, gini(S_2)$$

where, n_1 and n_2 are the number of data points in S_1 and S_2, respectively, and n is the number of data-points in S.

As it can be seen, the calculation of the gini index is the most important step in the node-splitting stage in a decision tree. Also, it can be trivially observed that the process could be time consuming, since, to be able to calculate the gini of one particular potential split value, all the records have to be considered in order to obtain the n_1, n_2 and all the p_j's for each S_1 and S_2. Since, it is of primary importance to calculate the gini at all the potential points, viz., all the distinct data points in the currect data-set for each attribute, any algorithm to do the gini calculations would necessarily take, $O(an^2)$ time complexity, where n is the number of records in the data-set and a is the number of attributes.

2.2 SLIQ Classifier for Data Mining

SLIQ was one of the first of its kind to introduce the concept of gini index to grow decision trees. The algorithm is divided into two phases, viz, **Tree Building** and **Tree Pruning**, for building decision diagrams. In the following sub-sections, these two stages are discussed.

2.2.1 Tree Building

This comprises two steps, i) *evaluation of splits for each attribute* and *selecting the best split* and ii) *creating of partitions using the best split.* This is done in the following manner. First the given table is split into separate lists, in which the records are sorted according to that particular attribute but maintain a pointer to the other attribute values of the same record, as can be seen below. The second column in each of the attribute lists are the record identifiers from the old (input) table, which maintain the class (dependent) attribute, as can be seen in the class list. The second column in the class list gives the current node that the particular record belongs to. Every node also stores a histogram or distribution of records (class values). Table 2-2 shows the original input table and Table 2-3 shows the separated *attribute lists* and the *class list*. The splits are evaluated using the EvaluateSplits() algorithm as shown in Figure 2-3.

Table 2.2: Sample dataset to demonstrate SLIQ

RID	Age	Salary	Class
1	23	15	G
2	30	40	B
3	40	60	G
4	45	65	B
5	55	75	G
6	55	100	G

Table 2.3: Attribute and class lists for SLIQ

Age List	
Age	CL Index
23	2
30	1
40	3
45	6
55	5
55	4

Salary List	
Salary	CL Index
15	2
40	4
60	6
65	1
75	3
100	5

Class List (CL)		
Index	Class	Leaf
1	G	N1
2	B	N1
3	G	N1
4	B	N1
5	G	N1
6	G	N1

Depending upon the best split value, the node name entry in the class list is updated according to the whether the record is pushed to the left or right sub-

```
Algorithm EvaluateSplits()
 for each attribute $A$ do
     traverse attribute list of $A$
     for each value $v$ in the attribute list do
         find entry in the class list, and hence the class and
         leaf node, $l$
         update the histogram (statistics) in leaf $l$
         if $A$ is a numeric attribute then
             compute splitting index for test $(A \leq v)$ for
             leaf $l$
         if $A$ is a categorical attribute then
             for each leaf of the tree do
                 find the subset of $A$ with the best split
```

Figure 2.3: EvaluateSplits()

tree, depending upon the best split condition $A \leq v$. The records satisfying the condition are placed in the left sub-tree, and others in the right sub-tree.[1]

2.2.2 Tree Pruning

The tree is built using the entire training data-set. This could contain some spurious "noisy" data, which could lead to a error in determining the class for the test data. Those branches that potentially could be misleading at run-time for class estimation are removed from the tree, using a pruning algorithm descibed in Mehta et al. [12].

2.3 SPRINT—A Parallel Classifier

SPRINT was one of the pioneering algorithms for building decision diagrams, that are **exact** classifiers as opposed to **approximate** classifiers, that compromised on accuracy for a better time complexity algorithm. Some of the approximate algoritms include C4.5 [7] and D-2. The algorithm was designed in such a way that it would be inherently parallel in nature, and hence leading to further scope for a speed-up as compared to the contemporary algorithms.

[1] This could vary depending upon the application using the decision tree. For a particular application the condition could be modified as $A < v$.

2.3.1 The SPRINT Algorithm

The algorithm at a macro-level is similar to its predecessor, SLIQ in that the algorithm comprises two parts, the tree growth stage and the tree pruning stage. The tree growth part of the algorithm has been modified to allow for parallelism, in the following manner. Like in the case of SLIQ, the given data-set is broken down into multiple tables—one for each attribute, and preserving pointer to the old record ID in the original data-set. The individual attribute lists are then sorted, before the histograms can be generated. Associated with every node is a histogram (distribution of the classes amongst the records in that node). A histogram is actually a matrix of possible class values and up/down distribution, implying number of records of the same classes lesser than or equal to (down) and greater than (up) the current record, under investigation. Using these histograms, the gini values for each attribute value can be calculated. Table 2-4 is a sample data-set and 2-5 shows the calculation of histograms at a few stages for a given attribute list.

Table 2.4: Attribute lists 'Age' for SPRINT

Age	Class	RID	Cursor Position
17	H	1	1
20	H	5	
23	H	0	
32	L	4	3
43	H	2	
68	L	3	6

Table 2.5: Attribute and class lists for SLIQ

		H	L
Cursor	C_{below}	1	0
Position 0	C_{above}	3	2

		H	L
Cursor	C_{below}	3	1
Position 3	C_{above}	1	1

		H	L
Cursor	C_{below}	4	2
Position 6	C_{above}	0	0

2.3.2 Speedup over SLIQ

SPRINT had a better speed-up over SLIQ, majorly owing to the fact that it had a better time complexity, $O(n \log n)$ as compared to the (O^2) as that of SLIQ. Another advantage that SPRINT had over SLIQ, was that, since only one attribute is needed to be processed at a time, only one list could be brought into memory at a time, as compared to SLIQ where the whole table (or corresponding parts of it) had to be memory resident to be able to calculate the best split. As a special case of SPRINT, if one whole attribute list could be made memory resident (at a time), along with the histogram (associated with that node) for storage of statistics, one can achieve a further speed-up obviating the necessity for swapping the list back and forth between memory and disk.

2.3.3 Exploiting Parallelism

The algorithm is inherently parallel in nature, whereby the entire data-set is converted to multiple attribute lists (smaller tables), that can be processed (gini calulation and determination of best split) in isolation, immaterial of the other attribute values for the same record. Thus, one can think of ways to parallelize the above algorithm, by assigning each attribute list to a separate processor for gini calculation, and then putting the result together to estimate the best split at a node. The node splitting can also be done in parallel in the following manner. Two processors enlist the subtree that each record should belong to after the split, depending upon the split attribute value. The attribute list being sorted, it is trivial to decide over the cut-off boundaries for each processor and hence they can work in parallel. Since, there can be no record common to either, they can both work on a **common** array in shared memory. This array, then can be used to split other attribute lists depending upon the entry in the shared memory array. Hence, splitting can be done in $O(n)$ time using $O(s)$ processors, where n is the number of record in a node and s is the number of attributes, hence preserving the total processor work, to $O(ns)$. This can be further extended to calculate the total spliting time complexity at the tree growth stage. Since there can be utmost $O(N)$ records in all the nodes at any level in the tree, the total time splitting time complexity of $O(N)$, where N is the total number of records in the data-set. Assuming a well distributed full tree, the total time complexity can be estimated to be $O(N \log N)$ for a $O(s)$ processor parallel machine.

2.4 CLOUDS—A Large Data-set Classifier

CLOUDS[2] was the first of its kind to use sampling for the purpose of classification. The sampling step was followed by an estimation step to determine a closer and better split attribute-value pair. The CLOUDS algorithm assumes the following two properties for gini indexes for real data-sets [11]:

- Given a sorted data-set, the gini value generally increases or decreases slowly. This implies that the number of good local minima is significantly less than the size of the data-set, especially for the best split attribute.

[2] Classification of Large or OUt-of-core DataSets.

- The minimun gini value (potential split) for an attribute is significantly lower than the other data-points along the same attribute and other attributes too.

Using these two principles as guidelines a couple of sampling techniques were developed:

2.4.1 Data-set Sampling (DS)

In this algorithm, a random sample of the data-set is obtained, and the direct method (DM)[3] for classification is applied. In order to maintain the quality of the classifier, the gini values are calculated using the entire data-set, only for the sampled data-points.

2.4.2 Sampling the Splitting Points (SS)

Here, a quantiling techique is used to partition the attribute domain into q parts. Gini values are calculated for each of the boundaries of the q-quantiles, and the lowest is chosen for the *split attribute*. Hence, it is required to have a pre-knowledge of the type and range of the attribute values (meta-data).

2.4.3 Sampling the Splitting Points with Estimation (SSE)

The SSE, technique uses SS to estimate the gini values at the boundaries of the q-quantiles for each attribute of the data-set. Then, as in the case of SS, the minimum $gini_{min}$ is chosen, here for the purpose of determining the threshold value for the next (estimation) set to determine the lowest gini value. Using the gini values, the lowest possible gini value in a quantile is determined, $gini_{low}$. Intervals that do not qualify the threshold level are discarded, i.e. intervals such that $gini_{low} \geq gini_{min}$, are eliminated. For the surviving intervals, gini values are calulated at every data point to determine the *lowest possible gini value.*

[3] Something like SPRINT, wherein the gini at every attribute value is calculated for estimating the best split.

CLOUDS uses both of the above to classify the data-set using sampling techniques. The sampling technique determines the size of the decision tree. The quantiling technique rules the accuracy rate and the time required at every stage.

2.5 Incremental Learners

The objective in having an incremental algorithm is that in case of the existing algorithms for building decision diagrams for new upcoming data, the entire classifier (learner) would need to be destroyed and a new learner created using the old and new data. Such a process would take a long time and would be repeated frequently. An incremental algorithm is such that the time required is corresponding merely to the new data, rather than the total of new and old data. One of the ways to achieve incrementality in the algorithm is, if one could have some technique to merge two learners together to obtain one learner that is a combination of the two learners. Chan and Stolfo [13, 14] have suggested some methods for merging trees together. The following are the two major techniques suggested:

- **Hypothesis booting** is a method in which a number of different algorithms are used on the same data-set to generate various learners. Then, using a meta-learner, all these various learners are combined. Thus, the properties of all the different learner algorithms are present in the new learner.

- **Parallel learning** is a technique in which a data-set is broken up into various parts, on which the same algorithm is applied to obtain different *parallel* learners, which can be combined together to obtain a learner for the whole data-set.

The other techniques comprise a combination of these ideas.

The following chapters give the Algorithms and the Implementation details of the Randomized Decision Tree algorithm along with performance statistics.

CHAPTER 3
ALGORITHMS

Having discussed the previous work in the area of classification data mining and specifically in the area of algorithms for decision trees, in the previous chapter, this chapter deals with the algorithms devised for the purpose of building (growing) randomized decision diagrams. The contemporary algorithms like SPRINT and SLIQ aim at building the most concise and compact form of the trees for a given data-set. But, as discussed before, this approach is exteremely time consuming. The present chapter discusses a few randomized algorithms that possibly could have the same time complexity, but are estimated to run faster, without a loss in accuracy in the outputted learner. In certain cases, the height and width of the tree are more than the SPRINT/SLIQ version of the tree for the same data-set.

The following sections give the drawbacks of the above contemporary algorithms that render them less useful for rapidly changing enormous amount of data or applications where the data could be outdated very early.

3.1 Sorting Is Evil

One of the most important characteristics of web-based applications is that the data is changing on a continuous basis and very little down time is permissible, if any. In an application, like *Click-stream Analysis*, it could be, in most of the cases, required to absorb and reflect a newly available data-set into the learner. In such cases, the decision tree should not be made from scratch but should be an addition to the already existing one. Hence, if it is required to sort the entire data-set for each attribute, the opertion will be extremely costly. Thus, the sorting operation that needs to be done (though only once) at the root node should be avoided as far as possible. If sorting cannot be avoided, then the number of records that have to be sorted should be reduced drastically.

Inspired by the SS approach as suggested in Alsabti et al. [11], the following algorithms suggest ways in which one can come up with decision diagrams without sorting the entire data-set.

3.2 Randomized Approach to Growing Decision Trees

Motwani and Raghavan [15], Horowithz et al. [16], Cormen et al. [17] and others suggest algorithms in which randomised approaches help in reducing the time complexity of an algorithm, without significant loss of accuracy and in most cases with 99% or higher accuracy.

One disadvantage with using randomized algorithms, as suggested before, is that though one does not lose out on accuracy, the resulting trees could be wider and longer – resulting in greater time to make a decision using this learner. If the tree growth process is not controlled, the trees could end up being skewed up, increasing the time complexity of the decision-making algorithms.

In applications where accuracy is of extereme importance, examples being those of high risk applications or life critical ones, it might not be feasible to use such algorithms. Examples of such are *disease diagnosis* or a learner that differentiates a poisonous mushroom from a non-poisonous one. But in such cases, if the learner assures 100% accuracy at the cost of higher search/decision time, randomized approaches could prove to be useful.

In the subsections that follow, randomized algorithms and their modification for building decision diagrams are suggested.

3.2.1 SSE Without Sorting

Sorting the attribute list is the most time consuming task in calculation of gini values before the node can be split. The attribute lists have to be seperately sorted as there can be no co-relation between the order of any two attributes in a data-set, the reason being that given a data-set with n attributes, $(n - 1)$ of them are *independent* attributes while 1 is a *dependent* attribute—referred to as the *class attribute*.

The understated algorithm would work perfectly, in one of the following scenarios :

• There are one or more attributes that are partially dependent on one or more other attributes, in that their values/order can be predicted based upon the value/order of other attributes or a combination thereof.

• If the application that uses the decision tree could tolerate faulty results some of the times. This is possible if the application uses the decision diagrams to predict a behavior of a non-life-threatening identity. It could also come of use in scenarios where the result is required urgently—a faulty one would not do harm to the application, but a timely procurement of a healthy result would certainly help.

• One has a certain amount of pre-knowledge of the data-set, in that, one can, after looking at a few data-points, make a fairly good guess of the nature of the neighboring points. An example would be of a data-set generated at a weather station. Looking at the temperatures of a few data-points, one can definitely make calculated guesses about the neighboring point (at least, one is sure that the night temperature is lower than the day temperature).

The algorithm proceeds with sampling a certain percentage of records and initially working with them. The gini values at these points are evaluated. Since, we will have a constant number of sampled points the complexity would necessarily be $O(n)$, where n is the total number if records in the data-set.

Here the gini values for the sampled records are calculated using the entire data-set, and hence these gini values are *exact* as opposed to *approximate*. This can be achieved in one of the following ways :

• Since we have only a constant number, s, of sampled records, one can obtain the statistics required for the gini calculation by merely comparing every record in the data-set with every record in the sampled set (records for which the gini is to be evaulated). This would require $O(ns)$ time or, if only a constant number of records are sampled, $O(n)$ time.

• If the number of records sampled is large, it could be costly to compare every one of the sampled records with the ones in the data-set. Here, we sort just the sampled records in $O(s \log s)$ time and then use the above process, in such a way that, if a record X lies ahead, in order, of another record Y for an attribute z, in the sampled set, then one can assume that for a record M in the data-set, if $M.z \leq X.z$, then $M.z \leq Y.z$ is also true. Thus, using techniques like BinarySearch or searching the array in the reverse order can help reduce the time required to determine the statistics.

Using the gini values for the sampled data-points, as in the case of SSE , the surviving intervals are selected. Here, unlike SSE since the sampled points have not been picked up from a pre-sorted data-set, one cannot guarantee the location of the ultimate minima. But, with a certain pre-knowledge about the data, like of the type mentioned above, it could be possible to figure out an approximate position of a local minima in an interval using techniques like BinarySearch to reduce the time spent in carrying out the search. As explained above, such a method would not yield the best of results and the trees could be larger,[1] but in cases, as discussed above, it could be worth having an algorithm that builds a larger tree in a shorter time frame.

3.2.2 Sampling a Large Number of Potential Split Points

In most of the contemporary databases, one does have a pre-knowledge about the data itself, in the form of meta-data (or data about data). One does know the domain of possible attribute values a particular attribute could have. It would prove advantageous to exploit this knowledge to build a classifier so that one can do the same, much faster. Now, note that a classifier is a data-structure, such that at every level, one makes a decision wherein one selects a path, one would traverse, depending upon a certain attribute value. The deciding factor is an attribute and the threshold value that determines whether to search (or continue traversal) in the left or right subtree. This threshold value is such that one gets the best possible tree, in that the decision be made as soon as possible, with no requirement that the value must exist in the training data-set (data-set required to grow the decision diagram). The data-point selection can be done in one of two ways explained below.

- If one has information about the data-set and the range of values each attribute could have, then the sampled data-points could be synthetically generated, so that they lie in the range covering all the possible values one can find in the data-set. Then, one could use the same method used in the algorithm described above to obtain a set of gini values for the selected data-points. Further techniques like searching for a lower interval in surviving intervals could also be exploited to zero on to the lowest (best) possible gini value, hence, determining the best split.

- Another technique one could use is that one samples a few records and

[1] longer and wider

uses only those as potential splits. This technique could be useful in cases where the data-set contains a lot of repeated data-points. In such cases, if a good sampling technique is used, one can expect the best split point to be sampled for gini calculation.

Depending upon the data-set one or more of the above methods could be used for sampling. If the range of possible values for an attribute is small and *discrete*, then it could prove to be advantageous to synthetically generate a large number of potential split points for that attribute. If the attribute values are *continuous* then one could use the method of sampling a percentage of the records for further calculation. Thus, depending upon the type of attribute, one could change the strategy being used for sampling a smaller set of potential split points.

3.2.3 Improvised Storage Structure

In SPRINT and the algorithms discussed so far, the data-set is converted to an intermediate representation, wherein, the attributes are split into various attribute lists that can be individually sorted. To preserve the records, the class list is created having incoming pointers from the individual attribute list, and stores the node that every record belongs to, at any stage in the algorithm. The advantage in having separate lists is that, one can just bring one list at a time in memory and process it in isolation (detached from the other parts of the record). But, with the algorithms stated above, this could imply a large number of comparisons and memory swap-in-swap-outs.

Thus, if one can have the whole records stored in-memory, before the comparison stage, all the comparisons required with a record, from the data-set, could be done at a time. Thus, it could be very convienent to compare the z-th attribute of the s-th record from the sampled set and the n-th record from the original data-set, for each z. A 3-dimensional array could be one such implementation.

Figure 3-1 suggests a method in which one can store the distribution statisics for each record. The three dimensions are of *records* (or records IDs), *attributes* and *classes*. The algorithm to populate the 3-D structure is shown in Figure 3-2 below.

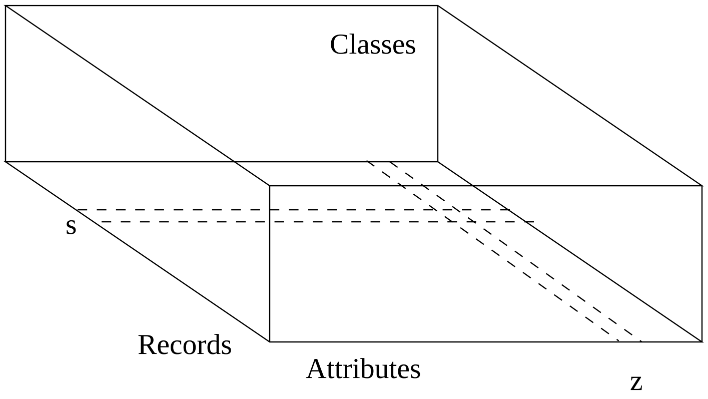


Figure 3.1: 3-dimensional array

```
Algorithm Populate3dArray
  Let the original data-set, $N$, contain $n$ records
  Sample $s$ records from $N$ to form the sampled set, $S$
  **for** each $n$ from $N$, belonging to current node **do**
      **for** each $s$ from $S$ **do**
          **for** each attribute $z$ **do**
              **if** $n.z \leq s.z$ **then**
                  Let $k$ be the class of record $n$
                  **increment** the $k$-th class-position of $s$
```

Figure 3.2: Algorithm Populate3dArray()

Using the statistics from the 3-dimensional array, one can calculate the gini indexes for each of the sampled record s. The advantage of having such a storage structure is that one can then pass on the information, if need be, from one stage to the other – more specifically from the parent node to one or each of its children. This reduces the time complexity by obviating the need to collect the statistics, everytime. But, this is not possible in the algorithm as is. This is because, while collecting statistics, we determine the number of records lesser than or equal to the current record's attribute value. The records, in the data-set, that we compare with are those that belong to the current node. In case of a split about one of the z attributes, it would be difficult to carry forward the statistics for all the attributes. One of the following algorithms suggests a method of storing the statistics in such a way that they could be carried forward to the child nodes.

3.2.4 Better Split-points

As it has been discussed before, it could prove to be advantageous to use attribute-values, other than the ones present in the data-set. The techniques that have been described before, use methods like using synthetic data-points for the evaluation of gini values. Other methods described in [10] and [9], use techniques like splitting an interval into two and using the median for gini calculation. Here, another technique is discussed that also aids the sampling techniques mentioned before.

The algorithm is necessarily similar to the ones mentioned before, that use sampling to obtain potential split points as opposed to calculating gini-values for all the data-points. Initially, a certain percentage of sampling is done. Then the sample set, S is increased three-fold by triplicating the records, in that, two data-points are inserted in every interval. This can be done in linear time, i.e. in $O(n)$ time one can find the thirds of every interval and designate them as data-point, as shown in Tables 3-1 and 3-2, below.

This can improve the performance of the previously stated algorithms in the following manner:

- Sampling of potential split points harms the decision diagrams in only one way – though the tree is not inaccurate, it could be wider and longer, reason being

Table 3.1: Sampled attribute values

Attribute value
2
3
6
8
14

Table 3.2: Sampled attribute values with interpolated values

Attribute value
2
2.33333
2.66666
3
4
5
6
6.66666
7.33334
8
10
12
14

that while sampling, the *best split attribute-value* could not have been sampled (and hence used to decide the best split). Using the present algorithm can help reduce the severity of this problem. As an example, in Figures 3-1 and 3-2, assuming that the best split is obtained at attribute value 12.1, which was not sampled. Using the technique of the thirds, a closer value, viz 12 was obtained, which could at times prove to be better than 12.1, in itself, in that, the resultant tree could be smaller.

- Using the method of the thirds, a closer sampling interval is obtained for a better granularity. This can aid in zeroing on the best split point, or the near best split point for algorithms like the ones descibed in [11] or a modification thereof, as suggested in section 3.2.1.

In the present algorithm and other randomized algorithms described above, the sampling technique reduces the number of gini calculations being performed, hence reducing the time required at every stage in the algorithm. Yet, the major bottle-neck, viz., collection of statistics of class distributions for gini calculations, remains to be cost in-efficient. The following sections address the issue.

3.2.5 Accelerated Collection of Statistics

In most of the algorithms used for building decision trees, every record is compared with every other record for collection of statistics used in gini calculations, with SPRINT as an exception. In the above randomized algorithms also, every record s from the sample set S is compared with every record n from the original data-set N. Since, the number of records in S is near constant, the complexity of the overall comparsion is $O(n)$, nearly linear. But, in scenarios, where a higher sampling is required, the performance would deteriorate. The following approach helps in reducing the number of comparisons.

The collection of statistics in the randomized algorithms above is done such that, the class dimension of every sampled record for each atrribute, gives the count of number of records in the original data-set that are less than or equal to the attribute-value, in question. Thus, for every attribute value j, for every record of the data-set, time is spent to find all the sampled records that have their attribute values greater than or equal to j. This could be avoided by using BinarySearch and then PrefixComputation to obtain the statistics in a manner described below. Before, the algorithm is stated, it is important to make the following observations:

• The attribute values when inserted in the 3-dimensional structure, mentioned above, are in sorted order. Hence, one can search for an attribute value in $O(\log n)$ time.

• Consider two records, A and B, that are sampled and stored in the 3-dimensional array, such that, for an attribute z, $A.z < B.z$ and hence A preceeds B in attribute list for z. If given that, for a record C belonging to the original data-set, $C.z < A.z$, the k-th location of the class dimension has to be incremented for *both* $A.z$ and $B.z$, where, k is the classification of C.

Thus, the algorithm to collect the statistics is as shown in Figure 3-3.

The statistics obtained thus are useful in the following manner:

• The class dimension does not give a direct measure of the number of records in N that are less than or equal to the current record, but a measure of number of records with their attribute values lying in the interval demarked by the current attribute value and the one prior to it in the 3-D array. This information could be useful in a more detailed search to zero in on the best split if need be.

• Using this information, the statistics of the number of records with attribute values less than or equal to the current record can be obtained using PrefixComputation. Calculation of prefixes being a purely mathematical operation, here addition,

```
Algorithm FastStats
  Let the original data-set, $N$, contain $n$ records
  Sample $s$ records from $N$ to form the sampled set, $S$
  for each attribute $z$ do
      Sort the $s$ records according to the $z$-th attribute
      Insert the $z$-th atrribute-values (only) for the sorted records in
      the 3-d array
  for each $n$ from $N$, belonging to current node do
      Let $k$ be the classification of $n$
      for each attribute $z$ do
          Use BinarySearch() to find the first records in $S$, such
          that $s.z \leq n.z$. Let it be $q$.
          Increment the $k$-th cell contents in the class dimension
          for $q$.
```

Figure 3.3: Algorithm FastStat()

can be done faster than record-comparisons, on any standard machine. Also, due to its inherent nature, PrefixComputation algorithm is parallelizable. Horowitz et al. [16] cite parallel-algorithms for computation of prefixes.

Putting it together, the techniques of sampling potential split data-points, the 3-dimensional storage structure for the sampled records and the class distribution and the accelerated collection of statistics for gini calculations can help in achieving a very high speed-up, at no loss of accuracy.

3.3 Multi-level Sampling

Unlike traditional databases, wherein no two records are identical (ideally), in the case of web data, there could be a lot of duplicate records. Infact, in many cases, the data could even be contradictory. Records could be contradictory, in a scenario, whereby given a data set with n attributes, $(n-1)$ of them being independent attributes and one dependent attribute or the *classification* of the record, if there exist two records, A and B, in the data-set, such that for A and B, all the $n-1$ independent attributes values match, but the dependent attribute does not. Thus, while generating the decision tree, either A or B or both would have to be

eliminated. This fact could be made use of, while sampling the data-set, such that a small number of records are used for classification.
The techniques used before ensure that the number of data-points, at which gini values are calculated, are a good sample of the data-set, such that the gini values are not calculated for duplicates. This improves the performance at the cost of the size of the tree, but does not affect the accuracy. The following algorithm is *approximate* in that outputted learned could produce inaccurate results for a few cases.

The algorithm proceeds with drawing out a random sample from the data-set. The percentage of sampling can vary according to the degree of inaccuracy tolerated. Using these sampled records and data-points, one can build a learner using any algorithm stated above. It can be argued that, due to the nature of the web data, the classifier would be fairly accurate, for a good sample of the data.

The following chapter comments on the accuracy of the algorithm that uses *two levels of sampling*, for building a classifier. The accuracy can be further improved by iterating through the process a fixed number of times using an *incremental algorithm* as described in the section 3.5.

3.4 "Say No to Randomization!"

Randomized algorithm for building decision diagrams can prove to be most beneficial to applications that would only benefit from a classification tool. Also, they can be very effective in applications where the tree needs to be re-constructed over and over again, frequently, over a small interval of time. In applications, where data generated due to web-clicks, could be misleading and inconsistent, to begin with, the classifier would only be as good as the data in itself. Hence, in such cases, using the two-level sampling technique to reduce the time required to build the tree, and an incremental algorithm, discussed in 3.5, to better the classifier, would be the best solution. But, randomized algorithms do have a few disadvantages and can be unacceptable in a few situations.

To list some of the disadvantages of the randomized algorithm for building classifiers -

- The method of selectively calculating the gini indexes of a few sampled data-points ensures that the the time complexity of the node split action is near $O(n \log n)$. But, it does not ensure that at every stage the best split attribute would be exploited. Resultingly, the trees could be much wider and longer than a traditional top-down algorithm that selects the best gini value at every node split, eg. SPRINT.

- In case of two-level sampling, the first level sampling, if not sufficient, could lose out on some non-trivial data-points, leading to a larger inaccuracy rate for the classifier at large. Thus, there is a trade-off between accuracy and time spent to build the decision tree.

Applications that have a high risk factor and are life threatening, could benefit little from such techniques, for the following reasons -

- The classifer for such applications would be expected up to have a very high accuracy rate, in absense of which, the application would produce faulty results.

- It could also be require to have a compact tree for the purpose of classification, so that the run-time to query on the tree is reduced. With longer trees the application would not be as beneficial[2].

Inspired by the techniques and data-structures used in the randomized algorithms, the following algorithms, use the complete data-set for the purpose of building decision diagrams, without randomization or sampling. The trees gen-

[2] This is theoretically true, although, as it can be observed in the chapter that follows, the length of the trees formed using a randomized algorithm are comparable with the most compact representation, and hence the run-times are also comparable

erated using the following algorithms are the most compact possible, because at every stage the best split value is selected.

3.4.1 Accelerated Collection of Statistics, the Reprise

This algorithm follows from the randomized version of FastStat algorithm. As described before, here the 3-dimensional array (storage structure) is used to store the class distributions prior to calculating the gini indexes. An algorithm similar to the one described before is used for the purpose of collection of statistics. Here, the records are not sampled. The entire data-set, in sorted order, is stored in the 3-dimesional structure. The complexity of the algorithm is hence, $O(n \log n)$, the time required to sort the entire data-set. But, this needs to be done just once, for the entire data-set. Unlike, the randomized methods, since all the records have been sorted once, one does not require to sort them again, at every node. An additional $O(n \log n)$ time is required at every node, for collection of statistics and populating the class-dimension. This is done using BinarySearch, as before, and then PrefixComputation is performed on them to obtain the class-distribution statistics. Since, operations happening at every stage are mostly mathematical, one can expect a speed-up over an algorithm that collects statistics by record comparisons.

Yet, since the node can split at any attribute-value, the statistics cannot be carried forward from a parent node to any of its children. The reason is that, the statistics give the class-distribution of number of records, belonging to that node, but less than or equal to the current record's attribute-value. Since the node can split at any attribute-value, the statistics, in their current format, cannot be carried forward from a parent node to any of its children. The following algorithm, stores the statistics in such a format that they can be passed over to one of the children nodes.

3.4.2 Statistics . . . To Go!

SPRINT can have a very high speed-up when parallelized. One interpretation of a parallelized version of SPRINT would be, assigning every attribute list to a processor that calculates the statistics for that attribute list and does the gini calculation. That is, the data-set is vertically fragmentable, to be processed in parallel. But, for every attribute list, the statistics are linearly incremented and hence it would be non-trivial to parallelize the data-set horizontally as well. The present algorithm, stores the statistics in such a way that the storage can be parallelized horizontally and vertically over the data-set. Also, they can be passed over to the child node, without loss of content.

The algorithm, for the sake of simplicity, assumes that all the values in an attribute list are unique. This assumption does not hurt the sequential version of the algorithm, but for the parallel version of the algorithm an extra step (compensating step) would need to be done to take care of duplicate elements. The algorithm proceeds as -

Sort each attribute list individually, according to the attribute values. Scanning the records sequentially, for every record j, the k-th location in the class-dimension is initialized to one, where k is the classification for that record. These are the *preliminary statistics* for the attribute list. Using these preliminary statistics, a prefix computation is done on all the records to get the *actual statistics* or *class-distribution.*

Since, the preliminary statistics are merely class-occurances of elements in the attribute list, when a node is split, these could certainly be passed over to one of the children nodes. At the child node, the algorithm can use the same preliminary statistics to obtain the actual statistics (class-distributions), using prefix computation.

For parallel version of the algorithm, the prefix computation can be done in parallel, using algorithms described in Horowitz et al. [16].

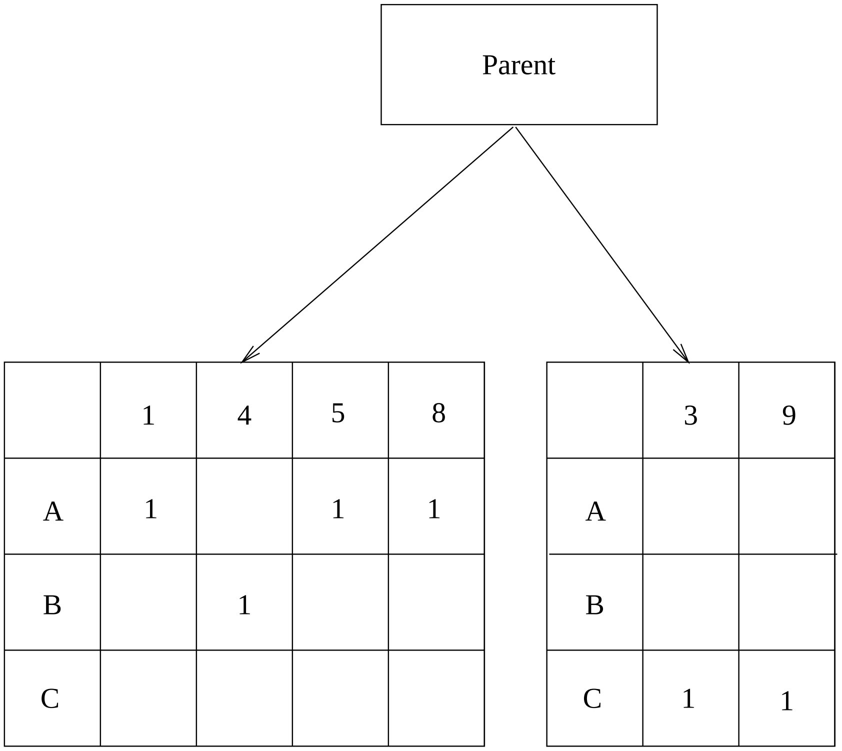
Figures 3-4 and 3-5 depicts preliminary statistics and actual statistics for an attribute, with no duplicates. Figure 3-6, shows the preliminary statistics being carried forward after the node-split.

	1	3	4	5	8	9
A	1			1	1	
B			1			
C		1				1

Figure 3.4: Preliminary Statistics for an attribute list - no duplicates

	1	3	4	5	8	9
A	1	1	1	2	3	3
B	0	0	1	1	1	1
C	0	1	1	1	1	2

Figure 3.5: Actual Statistics for an attribute list - no duplicates



	1	4	5	8
A	1		1	1
B		1		
C				

	3	9
A		
B		
C	1	1

Figure 3.6: Preliminary Statistics of children

To take care of duplicate attribute values, one of the following two methods could be used:

• While generating the premilinary statistics, each entry is treated to be unique, and the preliminary statistics are collected as before. Then, at the stage of prefix computation, the normal procedure is performed, but, for every unique attribute-value x, a pointer is maintained to the first occurance of x. No sooner the attribute-value changes, an equilizer function is applied on all the occurance of x, necessarily lying between the first and last occurance, thereof.[3] Then, one can proceed to a new value of x and repeat the process to obtain actual statistics.

• Another method, to solve the duplicate attribute-values problem is to have an extra *valid* bit for the 3-dimensional array for each entry in the attribute-record plane, i.e. each attribute-value, in the attribute list. During, the process of prefix

[3] Attribute lists are always maintained sorted.

computation, the valid bits for only the last occurance of ever attribute-value x are *enabled*, the other (prior) occurances are *disabled.* Only the *enabled* or *valid* attribute values are considered for gini-calculation.

However, while splitting the nodes, since only the preliminary statistics are passed over, the procedure remains unchanged.

The time complexity of the algorithm is necessarily $O(n \log n)$, but due to mere mathematical computations, at each level in the tree, the algorithm can be expected to have a better performance as compared to SPRINT. Also, the duplicate elimination technique mentioned above reduces the number of gini-calculations being performed, yet produces the most compact form of the tree.

3.5 Incremental Decision Trees

As discussed before, the need for an incremental algorithm is dire, in applications where new data is being generated at a high rate, and it is essential to use it in the process of decision making. In such a scenario, re-building a tree, periodically could be a solution, but, it has the certain drawbacks. If the most compact form of the tree is required, that is completely accurate (with respect to the training data-set), the tree building algorithm could be time consuming. In that case, there would be intervals of time, wherein the tree either would have the old data (uncommited to the decision-maker) or itself be unavailable for decision-making. To cope with the pressures of dire requirement for an incremental, algorithm that is continually available and is always accurate, the following algorithm could be used.

Consider a decision tree, T, having m levels and representing n records. The tree, T, is similar to the ones described before, barring that the leaf nodes hold pointers to the records that they represent. Let A be a new record that has to be inserted into the tree. The classification of A is c. To insert A into T, the tree is traversed starting at the root node, along the path depending upon the attribute values of A. At every stage, there could be one of two cases:

- A lands at a non-leaf node, with the split condition, attribute $j \leq x$. If $A.j \leq x$ traverse left subtree, else right subtree, subject to the condition that the left subtree satisfies the condition and right subtree falsifies it.

- A lands at the leaf node L, symbolizing class C. In this case, there could be two possibilities:

  ∘ Class of A, i.e. c conforms with the class of the node, viz. C. In this case, the record is dumped into pool of records embodied by L.

  ∘ Class of A, i.e. c does not conform with the class of the node, viz. C. In this case, the entire pool of records represented by L and A need to be put together in form of a tree. Any algorithm could be used at this stage, to build a tree using the already-existing records of the node and A. The root of this new tree, replaces L. In this case, the height of the tree could *possibly* increase by *one*.

The resultant tree represents $n + 1$ records, and has a height that satisfies,

$$m \leq height \leq m + 1$$

The above approach will serve as an incremental algorithm, but, as the number of records in the classifier increase, could prove to the highly inefficient. This is because, the tree increases in height at the leaf level, only, maintaining the same root node and other non-leaf nodes. Thus, once a node becomes a non-leaf node, it would remain there permanently. Thus, as an alternative, the entire data-set could be used to re-build a new tree T' when the number of records in the data-set, represented by T reaches 150% of its original value, or the record count crosses $1.5n$. One could also maintain the old tree T until T' has been created using the records held in the leaf nodes of T. It can be argued that such an approach could lead to the most compact tree structure frequently, while the tree predicts accurate results all the time.

A few algorithms described in this chapter, have been implemented. The implementation details and results are the subject of the next chapter.

CHAPTER 4
IMPLEMENTATION AND RESULTS

A few of the algorithms described in the previous chapter have been implemented. This chapter provides with the implementation details and the performance results. SPRINT is taken to be the benchmark for comparison.

4.1 Implementation

The present section describes the author's experience at implementing the algorithms. Java is selected as the language for implementation and the data sources are simple flat-files. In the subsections that follow, methods have been suggested for the use of alternative data sources. The datastructures, techniques and tools used for faster execution of the algorithms are discussed below.

4.1.1 Datastructures

The datastructures used for implementation of the algorithms have been defined in terms of generic re-useable java classes. A few of the native java classes have been used in some cases, without significantly affecting performance. *MyVector* class, that has better performance than Java's *Vector* class, has been defined to replace arrays in the algorithms. The structure and implementation details of *MyVector* class are a subject of section 4.1.6.

4.1.2 Implementing SPRINT

SPRINT is used as a benchmark of performance as well as accuracy – the *exact* randomized algorithms have been compared with SPRINT to test for performance and the *approximate* ones for accuracy. The comparison characteristics are given in section 4.2. For an accurate measure, SPRINT has been implemented in Java using the same generic datastructures, if needed, as the ones used for the randomized algorithms.

In incremental algorithms, for the case in which there is a disagreement between the new record and leaf-node class value, the data-points represented by the leaf-node and the new record have to be re-classified. Randomized algorithms cannot be used efficiently, because they tend have a reduced performance for a lower order data-set. The number of records contained in a leaf-node is of the order of a few hundreds, for a data-set with about 50000 tuples. Hence, the randomized algorithms are a worse option. Thus, for re-classification of the leaf-records, a SPRINT object is used.

Since, SPRINT is an exact classification algorithm, classification of test data obtained using randomized algorithms is tested using SPRINT.

4.1.3 Implementing the Randomized Algorithms

Random samples can be generated using one of the following methods, each has a complexity of $O(n)$ and can be used in different scenarios.

- One of method traverses the data-set once, completely. At every record, a coin is flipped – a random number between 0 and 100 is generated and is normalized by the sampling percentage to decide whether the record is to be sampled or not. This method proves to be useful, if one needs to scan through the data-set to collect information. The method also guarantees unique records in the sampled set. It can be used in cases like, determining the number of records in the entire data-set belonging to each class, wherein a complete prior-scan of the entire data-set is required.

- Another way to sample records is to generate a set, S, of the required number of records. Then for every $s \in S$, select the s-th record from the data-set.

This, method can be effective, where the data-set is memory resident, and there is no need to scan through the entire data-set.

In the implementation of the randomized algorithms, the data-set is scanned ones and stored in the form of arrays. At every stage in the algorithm, i.e. at every node, a fresh sample is generated, for the purpose of generating an un-biased tree. At deeper levels in the node, the number of records needed to be classified reduces, and hence only a small number of records need to be sampled, and it would be cost-ineffective, to scan though the data-set to select a very few records. Hence, the latter method is used. To ascertain sampling of merely unique records, a bit array is maintained, which is tested before selection of the set of random numbers (samples).

To maintain pointers to the data-points at the leaf-node level, the DecisionTreeNode class, extended class from the generic NodeBinary class has been defined, to aid in defining generalized object, usable by incremental algorithms.

4.1.4 Iterative as Opposed to Recursive

Java copies the parameters and objects across functions and scopes. Nested scopes produce duplicated data and the space occupied by it cannot be reclaimed unless the scope is exitted. Due to the nature of the algorithms for building decision diagrams, multiple nested scopes are generated for a recusive (easier) version of the algorithm. The algorithm would necessarily have to traverse the left-most path, before entering the right sub-tree. This, could cause memory to trash.

Both iterative and recusive algorithms have been implemented for SPRINT as well as the random decision tree generators. Iterative implementations tend to have lower execution speeds, but are optimized in memory usage, as the same data-store can be iterated through for different nodes. The decision tree nodes have to stored in an array format from iterative implementation while the recursive version inherently maps the recursive scope with the corresponding decision tree node. Section 4.1.6 suggests alternative store structure for the nodes in iterative code.

4.1.5 Alternative Data Sources

The current implementation of the decision tree making algorithms use flat-files as the source of data. The data from the source is extracted into a *DataObject* before the data is processed. This architecture enables the possibilty of raw data being fed from a new data source, with minor modifications to the data retrival module. The data is sorted (processed), re-organized in this data-object. Figure 4-1 details the architecture that makes plug-in for the new data source possible.

4.1.6 Embedded Server for Run-time Statistics

The time required to classify a data-set comprising tens of thousands of records on a SUN Ultra 10 machine is of the order of tens of minutes. To regulate the performance at every stage in the algorithm and generate run-time statistics, an embedded server is used. The server lies close to core of the decision tree builder and has access to statistics as the tree is built. The server can be queried upon by an external client, at run-time, for latest statistics. Figure 4-2 detail the architecture.

4.1.7 MyVector Class—for Better Array Management

Like the algorithms for building decision trees, arrays are a predominant storage medium for most implementations. The usage of arrays makes data retrival and storage faster and easier. But, arrays have the following disadvantages:

- The array bounds have to be fixed judiciously and then managed on a

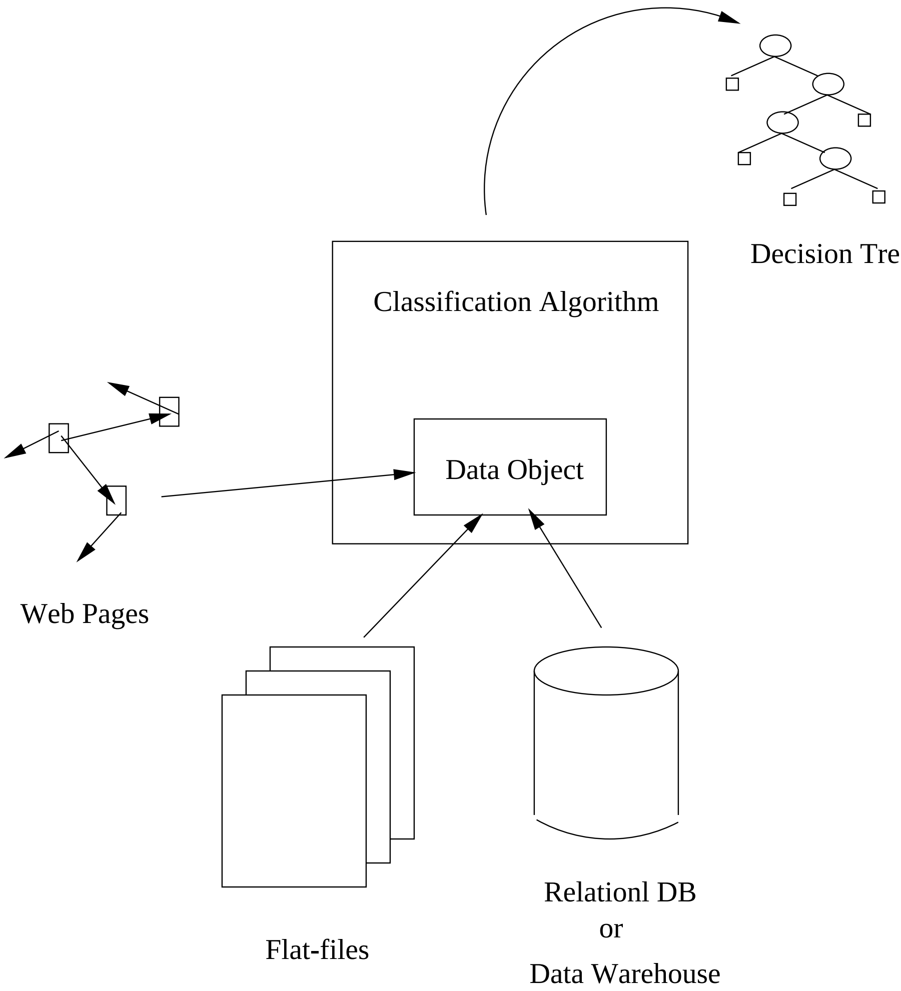


Figure 4.1: Alternative data sources

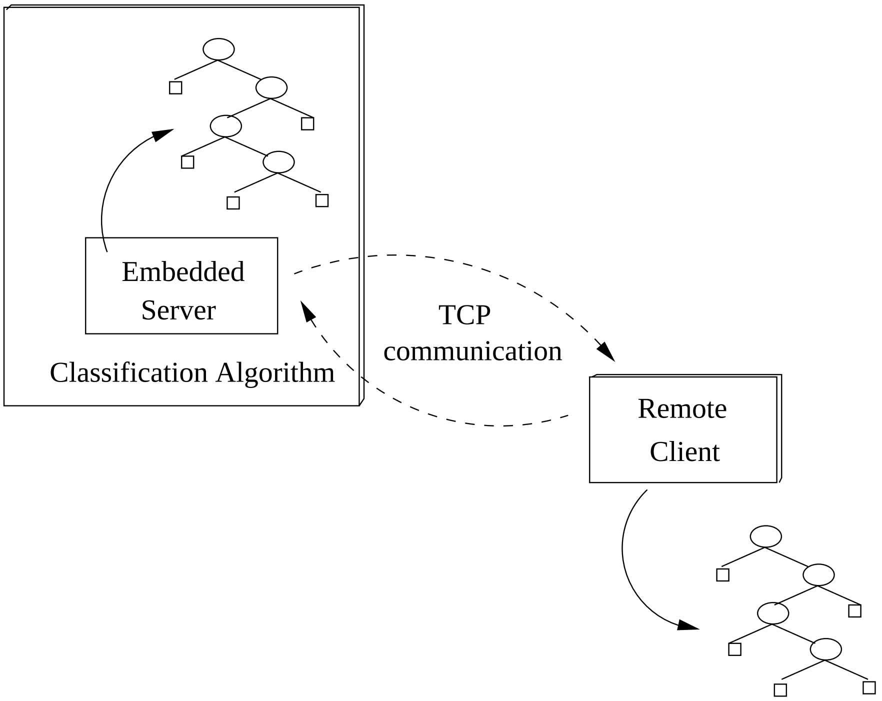


Figure 4.2: Embedded Server – Remote Client architecture

continual basis. Failure to do so can result in unpredictable results.

- When the array is needed to be expanded, dynamically, the array needs to be recreated to an alternative location and the already existing data has to be copied. This poses an extra overhead for array management.

To do away with the above disadvantages, MyVector class is defined, that uses arrays for internal storage, in form of blocks. The storage can be made extendible by adding blocks to the current store, making space for new data, without having to move the old one. Thus, it does away with the overhead of managing bounds and having to copy data for extendible storages. Figure 4-3 depicts the architecure of MyVector class objects.

Java provides a Vector class that does away with the overhead of having to manage the bounds. MyVector class however tends to have a better performance as compared to Vector class.

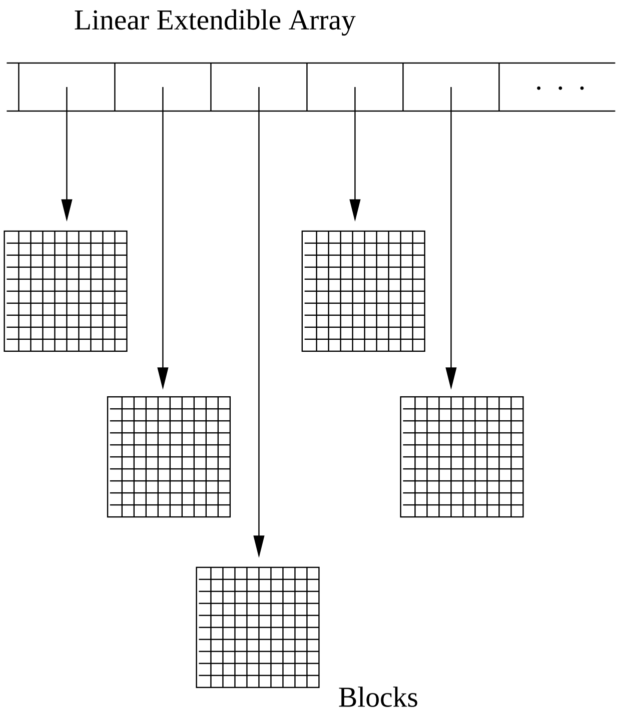


Figure 4.3: Architecture of MyVector class objects

4.2 Results

The section reports the performance results for the implemented algorithms. Various tests were run to compare the performance and accuracy of the randomized algorithms. The tests were done on *eclipse*, a SUN-Sparc 8 processor machine, on Solaris 5.6.

Majorly two types of test were run - performance tests for speed and accuracy tests for prediction reliability.

4.2.1 Performance

A randomized algorithm is expected to perform better (in terms of time required for exection) as compared to a sequential (non-randomized) algorithm.

One of the randomized algorithms, discussed before, viz., accelarated collection of statistics using binary search and prefix computation, is compared against SPRINT. The tests were run on the machine described above, with a data-set of

43500 records, 9 independent attributes and 1 dependent attribute (the classification). The tests were run, at various levels of first- and second-level sampling.

Figure 4-4, plots the two-level sampling results against, time. As can be seen, it out-performs SPRINT by a large margin. The algorithm more-or-less scales linearly.

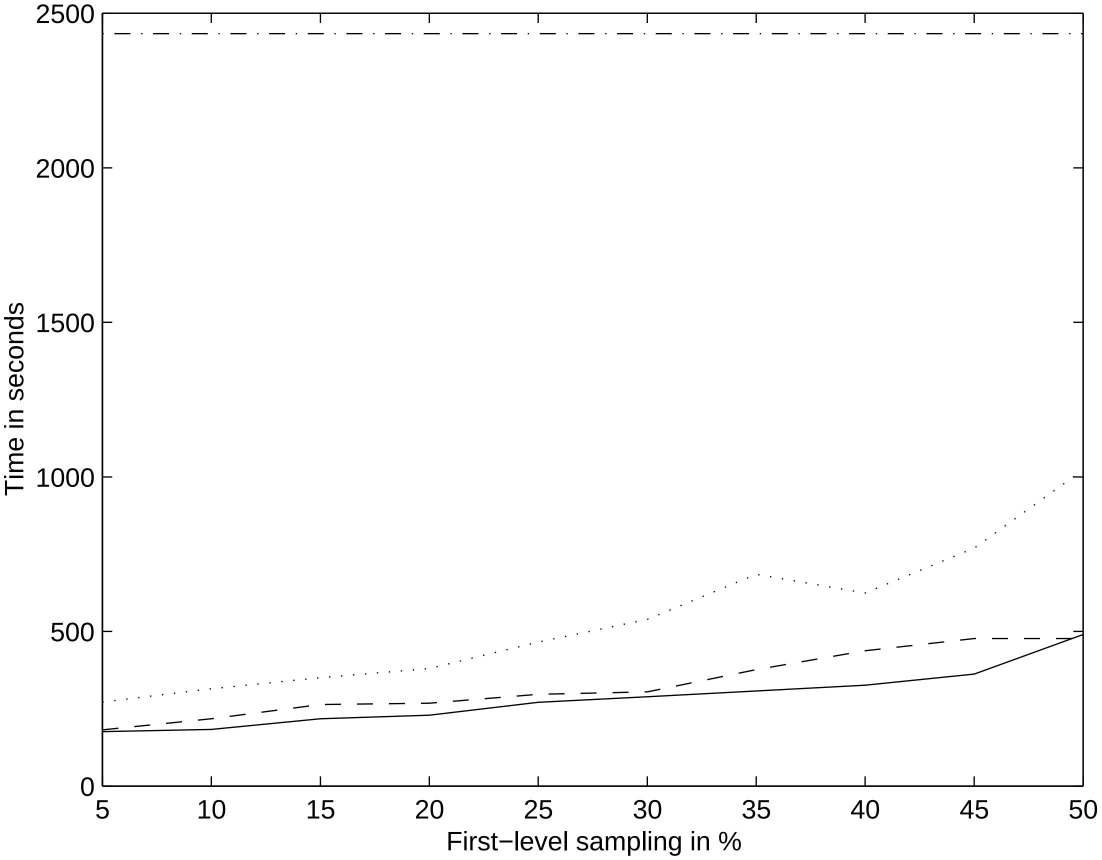


Figure 4.4: Performance of the Two-level sampling algorithm. Legend: Dot-dashed line (SPRINT), Dotted-line (5%), Dashed-line (2%) and Un-cut line (1% second level sampling)

Another test of performance comparison, done is between two randomized algorithms—the potential split points algorithms against the accelarated collection of statistics algorithm. As expected, the latter performs better than the first, due to the usages of techniques like Binary Search and Prefix Computation. Table 4-1 summarizes the results.

Table 4.1: Sample dataset compare performance of two randmized algorithms. Data-set of 846 records, 70% first-level and 20% second-level sampling

Iteration	Potential Split Points Algo. Algorithm	Accelerate Collection of Statistics Algorithm
1	54	20
2	56	17
3	54	11
4	53	15
avg. time in secs.	**54.25**	**15.75**

4.2.2 Accuracy

In case of the two-level sampling algorithm, since only a part of the data-set is used for generating the tree, it is only approximate. The predictions done using such a tree could be wrong, if the selected data is not well distributed. The algorithm was tested using small and large data-sets, at various first-level sampling. As can be seen in Figure 4-5, a 50% or higher of first-level sampling provide good results, in a lower time frame, as seen in Figure 4-4.

As can be seen above, with a lower first-level sampling, the algorithm has a high performance, i.e. takes lesser time to execute, but has a lower accuracy rate. With a high first-level sampling, the accuracy improves, but the run-time increases (though almost linearly). Thus, with respect to the application that uses the decision tree, one can come up with a fair amount of first-level sampling, that provides with acceptable accuracy and caters to the performances needs, as well.

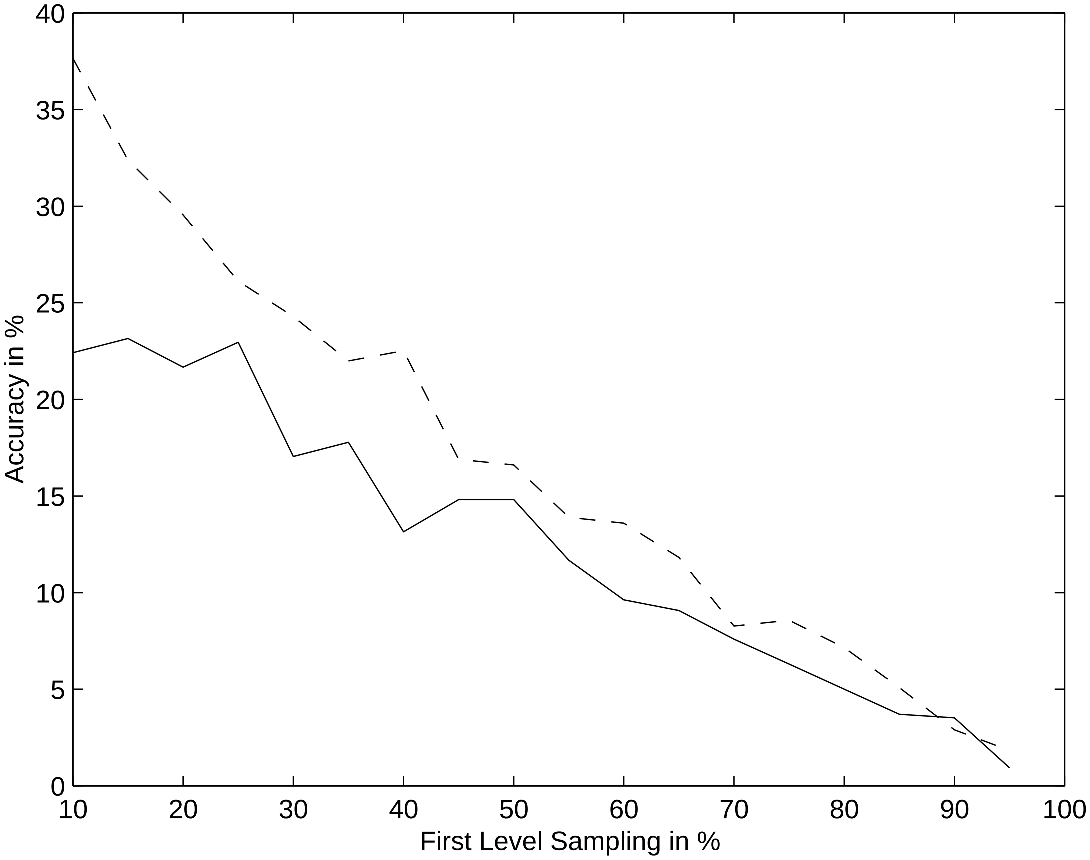


Figure 4.5: Accuracy of the Two-level sampling algorithm for two different data-sets: Un-cut line, data-set with 15000 records and dashed-line, a data-set with 846 records

CHAPTER 5
CONCLUSIONS AND FUTURE WORK

The current work discusses the need for a decision-makers, for web- and other applications. In some cases, the need for the model to be an exact embodiment of the input data-set or the training set, is dire. While, in the case of others, like click-stream analysis, a fairly good prediction made, at run-time, using the model, can help the application or boost up profits. The other requirement is that of the time required to build the model and that required to run a query on it. Depending on the application in use, one or both are needed to be optimized –a decision made while choosing an algorithm used to build the tree.

Randomized algorithms for building decision diagrams have been discussed. These have varying time-complexities, static build-time and dynamic query-time and accuracy rates. For life critical applications, an exact classifier would be required that has an optimized run-time, while for a business application an algorithm that build trees in the smallest possible time frame at a slight expense of accuracy could be desired/acceptable. In cases, where the data itself is inaccurate, one could profit with an algorithm like the latter.

Incremental algorithms are required in scenarios where the data continuously flows in and it is required to reflect the changes, if any, to the model at the earliest. The incremental algorithm, discussed here, optimizes on both the static and dynamic time and yet incremental in nature –achieving the best of both worlds.

Thus, using the set of algorithms discussed, most of the applications can benefit –achieving in much smaller time, almost the same result as an exact classifier would produce.

REFERENCES

[1] Sam Anahory and Dennis Murray. *Data Warehousing in the Real World*. Addison-Wesley, Reading, Mass., 1997.

[2] Jennifer Widom. Research Problems in Data Warehousing. In *Proc. of 4th Int'l Conference on Information and Knowledge Management (CIKM-95)*, Baltimore, Maryland, November 1995. (Invited paper).

[3] Rakesh Agarwal, Manish Mehta, Ramakrishnan Srikant, Andreas Arning, and Toni Bollinger. The Quest Data Mining System. In *Proc. of the 2nd Int'l Conference on Knowledge Discovery in Databases and Data Mining*, Portland, Oregon, August 1996.

[4] Rakesh Agarwal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules. In *Proc. of the 20th VLDB Conference*, Santiago, Chile, 1994.

[5] Ramakrishanan Srikant and Rakesh Agarwal. Mining Quantitative Association Rules in Large Relational Tables. In *Proc. of the ACM-SIGMOD 1996 Conference on Management of Data*, Montreal, Canada, June 1996.

[6] Rakesh Agarwal and Ramakrishnan Srikant. Mining Sequential Patterns. In *Proc. of the 11th Int'l Conference on Data Engineering*, Taipei, Taiwan, March 1995.

[7] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, California, 1993.

[8] J. Wirth and J. Catlett. Experiments on the Costs and Benefits of Windowing in ID3. In *5th Int'l Conference on Machine Learning*, pages 87-99, Ann Arbor, Michigan, June 1988.

[9] Manish Mehta, Rakesh Agarwal, and Jorma Rissanen. SLIQ: A Fast Scalable Classfier for Data Mining. In *Proc. of the Fifth Int'l Conference on Extending Database Technology (EDBT)*, Avignon, France, March 1996.

[10] John Shafer, Rakesh Agarwal, and Manish Mehta. SPRINT: A Scalable Parallel Classifier for Data Mining. In *Proc. of the 22nd Int'l Conference on Very Large Databases*, Bombay, India, September 1996.

[11] Khaled Alsabti, Sanjay Ranka, and Vineet Singh. CLOUDS: A Decision Tree Classifier for Large Datasets. In *4th Int'l Conference on Knowledge Discovery and Data Mining (KDD-98)*, New York City, August 1998.

[12] Manish Mehta, Jorma Rissanen, and Rakesh Agarwal. MDL-based Decision Tree Pruning. In *Proc. of the 1st Int'l Conference on Knowledge Discovery in Databases and Data Mining*, Montreal, Canada, August 1995.

[13] Philip K. Chan and Salvatore J. Stolfo. Experiments on multistrategy learning by metalearning. In *Proc. 2nd Int'l. Conference on Information and Knowledge Management (CIKM-93)*, pages 314-323, Washington, November 1993.

[14] Philip K. Chan and Salvatore J. Stolfo. Meta-learning for multistrategy and parallel learning. In *Proc. Second Intl. Workshop on Multistrategy Learning (MSL-93)*, pages 150-165, Harpers Ferry, Virginia, May 1993.

[15] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms.* Cambridge University Press, New York, 1995.

[16] Ellis Horowitz, Sartaj Sahni, and Sanguthevar Rajasekaran. *Computer Algorithms.* W.H. Freeman and Company, New York, 1997.

[17] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, Mass., 1990.

BIOGRAPHICAL SKETCH

Vidyamani Parkhe was born on April 21st, 1976, in Indore, India. He completed his bachelor's degree in electrical engineering at the Govt. College of Engineering, Pune, India, in June 1998. He worked as an intern as a developement engineer at the Loudspeaker Developement Lab., Philips Sound Systems, Pimpri, India.

He joined the University of Florida, in August of 1998. He worked as a research and teaching assistant for several courses. He completed his Master of Science degree in computer engineering at the University of Florida, Gainesville in December, 2000.

His research interests include randomized algorithms, data structures, databases and data mining.