A Lazy Model-Based Approach to On-Line Classification

by

Gabor Melli
B.Sc., University of British Columbia, 1989

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
April 1998

Abstract

The growing access to large amounts of structured observations allows for more opportunistic uses of this data. An example of this, is the prediction of an event's class membership based on a database of observations. When these predictions are supported by a high-level representation, we refer to these as knowledge based on-line classification tasks. Two common types of algorithms from machine learning research that may be applied to on-line classification tasks make use of either lazy instance-based (k-NN,IB1) or eager model-based (C4.5,CN2) approaches. Neither approach, however, appears to provide a complete solution for these tasks.

This thesis proposes a lazy model-based algorithm, named DBPredictor, that is suited to knowledge based on-line classification tasks. The algorithm uses a greedy top-down search to locate a probabilistic IF-THEN rule that will classify the given event. Empirical investigation validates this match. DBPredictor is shown to be as accurate as IB1 and C4.5 against general datasets. Its accuracy however, is more robust to irrelevant attributes than IB1, and more robust to underspecified events than C4.5. Finally, DBPredictor is shown to solve a significant number of classification requests before C4.5 can satisfy its first request.

These performance characteristics, along with the algorithm's ability to avoid discretization of numerical attributes and its ability to be tightly-coupled with a relational database, suggests that DBPredictor is an appropriate algorithm for knowledge based on-line classification tasks.

Acknowledgments

My thanks go to all the people whose efforts have allowed me to get to this point. I am indebted to you, and to show my appreciation, I will try my best to carry forward your spirit of giving.

Academically, I feel very privileged to have had Dr. Jiawei Han as my supervisor. His support and patience saw me through to the end and I will never forget the day he proposed "some topics"!

I am grateful for the support I received from the School of Computing Science, the Centre for Systems Science, and the Office of the Dean of Graduate Studies. Kersti Jaager and Dr. Stella Atkins, were particularly helpful with my last bureaucratic hurdles.

During this thesis I met and became partners with Maria Lantin. Her help and companionship resulted in a better thesis and has left me with a better appreciation for life. Similarly, my thanks go to my grandmother, whom this thesis is dedicated to, and my brother, Alexander Schmidt. The joy we have shared has given me the security to pursue my ambitions. Finally, I will always be indebted to my mother, Heidemarie Schmidt, for allowing me to create my own script, which includes the creation of this thesis, and providing me with the opportunity to pursue it.

Dedication

To my Oma, Marianne Schmidt
and my Opa, Artur Schmidt
...
Thank you for your love
and the precious memories of my youth

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Classification is an essential activity of all living organisms. As soon as we have sensed an experience, we need to quickly predict the class of this particular event. Important classes include edible, poisonous, hot, cold, friend, and foe. Beyond our genetically enabled abilities, human beings can also learn further classifications skills by way of instruction or observation. Similarly, computer programs can be developed to classify events based on instruction or observation. Expert Systems are examples of classifiers that have been instructed about a particular domain. Machine Learning systems, on the other hand, are examples of classifiers that can induce predictions about a domain, based on a set of observations from this domain. This thesis investigates a specific task within the latter approach: *knowledge based on-line classification* tasks. Consider the following example:

Example 1.1. A person is about to eat a wild mushroom but wants to confirm whether it is edible or poisonous. After a quick search, the person finds a very large SQL-based relational database on the World Wide Web with mushroom records composed of observable features such as edibility, weight, size, shape, and spore colour. The person would like to predict the edibility of their particular mushroom based on this set of observations. Because the dataset may be quite large and of suspect quality, the person would like an automated classification that supports its prediction with an understandable abstraction such as

IF Spores=purple AND Surface=warts AND Height $\in$ [3.3,4.5]cm
THEN Edibility=poisonous (71%) OR Edibility=edible (29%)
(based on 33 matching records)

With this rule the person has supporting evidence that their mushroom is edible, but that their is a strong chance that it is poisonous. □

The example is a knowledge based on-line classification task because it requires the prediction of a single event's class based on a database of stored observations and also required that the prediction be supported by a high-level rationale. For conciseness we will occasionally drop the *knowledge-based* description and refer to these tasks as *on-line classification* tasks. It is assumed that an understandable justifications however, remains an important requirement.

With the continued growth in database usage, on-line classification tasks will likely become more common in the near future. Many domains already collect a significant amount of data to support their day-to-day activities. This growth in data collection is made possible, by the general increase in capacity (disk, memory) and speed of computers. Because this growth in the number of database and our ability to process their information is expected to continue into the near future, some of these databases will likely support classification tasks for their specific domains. Already in the last five years, large databases, now known as "data warehouses" are being used extensively for *On-line Analytical Processing* (OLAP) tasks [15]. OLAP tools not only helps an expert analyze large sets of records, but also allows more sophisticated analysis by an expanded group of people. As more domains are captured within databases, and computational power increases, better tools may also allow on-line classification tasks to become more common place.

This thesis proposes an algorithm, named DBPredictor, that is targeted at on-line classification tasks. While significant investigation has already been performed on general classification, the two common approaches of *eager model-based* (C4.5 [58], CN2 [13]) and *lazy instance-based* (k-NN [17], IB1 [6]) classification do not provide a complete solution for on-line classification tasks. DBPredictor can be said to use, a *lazy model-based* approach which allows it to quickly return an accurate prediction that is also supported by a simple IF-THEN rule. Empirical investigations show that indeed, DBPredictor presents advantages over eager model-based algorithms in terms of speed and accuracy, as well as advantages over lazy instance-based algorithms in terms of accuracy and result understandability.

1.1 Motivations

Several issues motivate continued research into classification and specifically into knowledge based on-line classification. Recently, for example, the field of data mining, has arisen as a practical way to automate classification. Significant value has already been achieved in public interest domains such as medical diagnosis [45] and star classification [24], and in private interest domains such as financial markets [7] and automobile repair [67]. The classification tasks listed above occur against relatively well known domains where data is collected for the specific purpose of classification.

As confidence grows in classification algorithms and more domains have ready access to large amounts of information, future classification tasks will likely expand into less structured applications. Two such settings include opportunistic uses and dynamic domains. Opportunistic classification occurs against a dataset that was not collected specifically for the classification task at hand [36]. In the mushroom example, edibility was of interest. To someone else, the classification of the mushroom's habitat may have been of greater interest. As databases become more widely available, opportunistic classification requests may become common place. A different type of domain that may benefit from on-line classification, are those in which the underlying model periodically changes. Dynamic environments such as weather systems, stock markets and natural disaster relief occasionally require predictions based on very recent observations. In many ways, no weather system or stock market crash is like the next.

Opportunistic classification and classification in dynamic domains may gauge a classification algorithm's performance in slightly different ways than structured classification tasks. They will likely require that an understandable justification be presented. In this way, the person can involve their background knowledge in assessing the confidence they would place on a particular prediction. The accuracy of algorithms that serve these tasks should also be robust to the presence of irrelevant attributes and to underspecified event descriptions. In the mushroom example, it is likely that a person may enter information of insignificant predictive value, and may also be unable to produce key information about the mushroom.

1.2 Approach

Several fields have participated in the investigation of classification: this include the fields of Statistical Data Analysis [20, 34], Pattern Recognition [28, 37], Machine Learning [10, 62, 14] and more recently in Data Mining [27, 54]. Each of these fields have made significant contributions to the topic. Some approaches have proved to be more accurate in particular domains, others have well understood theoretical foundations, while others attempt to be compatible with human reasoning. Because knowledge based on-line classification tasks require database interaction and a generally understandable solution, this thesis draws mainly from the field of data mining and machine learning. Within the field of data mining, we draw from research into prediction [27]. DBPredictor derives its name from this association. Within the field of machine learning, we draw from the research into supervised learning from examples [19].

Two distinct and well developed approaches that may be applied to on-line classification tasks, include *eager model-based* and *lazy instance-based* classification algorithms [4, 59]. Eager approaches learn (induce) a complete classification structure before any classification requests may be processed. Lazy approaches forgo the learning phase and return a result tailored to the classification of the event at hand [5, 4]. Model-based approaches represent their result in a language that is richer than the language used to describe the dataset. Instance-based approaches represent their result in the same language that is used to described the dataset [59].

Eager model-based approaches include decision tree and rule induction algorithms such as C4.5 [58] and CN2 [13]. These algorithms have two distinct phases. The first, eagerly induces a high-level structure (decision trees or rules) and the second, classifies any new event based on this structure. Lazy instance-based approaches, such as the k-nearest neighbour based IB1 [6] algorithm, on the other hand, lazily postpones any work until an event that is to be classified is presented, then it quickly locates the instances that are most similar to the event and base its prediction on these instances.

These two approaches, however, encounter some difficulty when applied to on-line classification tasks. Eager model-based approaches expend a significant amount of time by returning a result that is unnecessarily general. In the mushroom example, the C4.5 and the CN2 algorithms would develop a structure that is capable of classifying **any** mushroom, not just the specific mushroom in question. The problem with lazy instance-based

approaches is that they return a result in a low-level representation that is difficult to interpret, and their accuracy is also susceptible to the presence of irrelevant attributes. In the mushroom example, the IB1 algorithm would return a set of records (instances) of other mushrooms that are most similar to the mushroom in question. These instances, however, may be related to the mushroom at hand on very irrelevant attributes and the person who sees the result, would not be able to easily determine whether this is the case.

Instead of using an eager model-based approach or a lazy instance-based approach for on-line classification tasks, a lazy model-based approach may be more suitable. A lazy model-based classification algorithm will restrict its efforts to the classification task at hand to expeditiously achieve a prediction. Its prediction will also be supported with a high-level representation. Two recent proposals that implicitly use a lazy model-based approach include the LazyDT [29] and DBPredictor [46] algorithms. The main difference between these two is LazyDT's use of a decision tree path representation and the DBPredictor's `IF-THEN` rule-based representation. Current implementations of these two algorithms however, are not particularly suited to on-line classification because of their memory intensive definitions, their requirement of discretized datasets, and their inaccuracy due to overfitting.

1.3 Contributions

This thesis describes the lazy model-based DBPredictor algorithm in more detail than in [46], proposes some enhancements to better support on-line classification tasks, and empirically validates its applicability to these tasks. The main contributions of this thesis are the following:

1. Natural handling of numerical attributes that removes the requirement for global discretization. The impact of this approach on accuracy is also empirically validated.

2. The addition of pruning and the empirical validation of its positive effect on accuracy and against overfitting.

3. The ability to directly interact with an SQL-based dataset.

4. An empirical demonstration that the algorithm is more accurate than C4.5 in domains with underspecified event descriptions.

5. An empirical demonstration that the algorithm is more accurate than IB1 in the presence of irrelevant attributes.

6. Identification of an appropriate heuristic function and the rejection of the parent-child approach to this function.

7. An empirical demonstration that the algorithm performs significantly faster than the C4.5 algorithm.

8. Support for concept hierarchies.

1.4 Thesis Outline

The general problem of knowledge based on-line classification and the motivation for further research in this area have been presented. The remainder of this thesis describes in the detail a framework for these tasks, surveys several current algorithms, proposes a new algorithm and then empirically validates the algorithm's value. These topics are grouped into nine chapters. Chapter 2 presents the framework for knowledge based on-line classification. Chapter 3 describes the possible application of several current classification algorithms to on-line classification tasks. Chapter 4 describes in detail the proposed DBPredictor algorithm and also presents a complexity analysis. Chapter 5 presents a faster version of the algorithm that requires significantly greater space resources. The purpose of this version was to facilitate the empirical study. Chapter 6 concludes the description of DBPredictor with an investigation of several heuristic functions. Chapter 7 presents the results of the empirical investigations into DBPredictor's accuracy characteristics, while Chapter 8 presents the results of the empirical study into the algorithm's running time characteristics. Chapter 9 concludes the thesis with a summary of the contributions and with some suggested directions for future research.

Chapter 2

General Framework

A knowledge based on-line classification task is a request for a prediction of an event's class that is based on a dataset from the same domain as the event, and that is supported by a high-level representation. This chapter presents a framework of these tasks to help determine whether an algorithm meets the requirements of such tasks. The framework also details how the performance of these algorithms will be measured. The chapter concludes with a review of several prediction task requirements that are closely related to on-line classification, but that are outside the scope of this framework. Where possible, the terminology in this chapter was drawn from previous studies of classification [12, 22, 42, 52, 58].

Section 2.1 presents an example that will help to tie in the discussions within this and other chapters in this thesis. Sections 2.2, 2.3, 2.4 describe the input, output and control requirements for these tasks. The focus will be on mandatory requirements, however some of the more common optional requirements are also presented. Section 2.5 describes the measures that will gauge the performance of algorithms attempting on-line classification tasks. Finally, Section 2.6 describes three areas that related to, but outside the scope of, on-line classification: regression, batch classification and system guided classification.

2.1 Example

To facilitate the discussion within this thesis many of its examples will be related to the sample Animal Kingdom domain that is described below. In this reference example there exists a set of records about many different animals that someone had taken the trouble to compile. There may be many entries for any given type of animal, but it is assumed that

there is only one entry for any specific instance of an animal. Based on this information another person may want to predict the value for a particular feature of an animal that was recently observed. A specific prediction request will provide some information that is known about the animal in question and will indicate which feature of this animal is to be predicted. Two sample requests are presented:

1. A person recently observed a small animal that ate berries and then flew away. The person recorded some information about this animal: surface covered in feathers, diet contained fruit, and size was small. This person may now want to predict the animal's family within the animal kingdom. A potential output to this request is that the most likely type of animal is a bird.

2. Another person may have instead encountered a large, heavy, four footed brown-coloured animal with antlers. Instead of the animal's family however, this person wants to predict the type of surface with which this animal is covered (e.g. feathers). Possibly the person is fearful to get too close to this animal to directly determine this information. A potential response to this query is that based on the current set of known animals, all known large antlered animals are covered with hair.

This sample domain will be developed more fully as more concepts about on-line classification are developed throughout this chapter.

2.2 Input Requirements

An on-line classifications task must provide three pieces of information before a classification algorithm can proceed with the request: $\vec{e}$, D, $\vec{c}$. The *event vector* $\vec{e}$ contains the information about the event whose class will be predicted. The *class attribute* $\vec{c}$ describes the attribute whose value (class) is to be predicted. The *dataset* D contains the information about the domain of the event vector that can be used by a classification algorithm to base its prediction.

2.2.1 Dataset (D)

Parameter D points to the dataset which contains records with information about the particular *domain* in question. For our reference example the domain is the Animal Kingdom and the sample data set is located in Table 2.1. A dataset is assumed to be composed of n *records* and m *attributes*. In Table 2.1, thirteen records are visible from a total of ten thousand records ($n = 10,000$). Each record represents a particular instance from the domain and is described by m *attribute-values*. These values represent either an empirical or derived feature of each instance. In the sample dataset there are seven ($m = 7$) attributes.

Table 2.1: Dataset example of the Animal Kingdom domain (based on Hu, 1992 [35]). The row identifier column exists for reference purposes and the ↓ symbol indicates that the rows are ordered on this column.

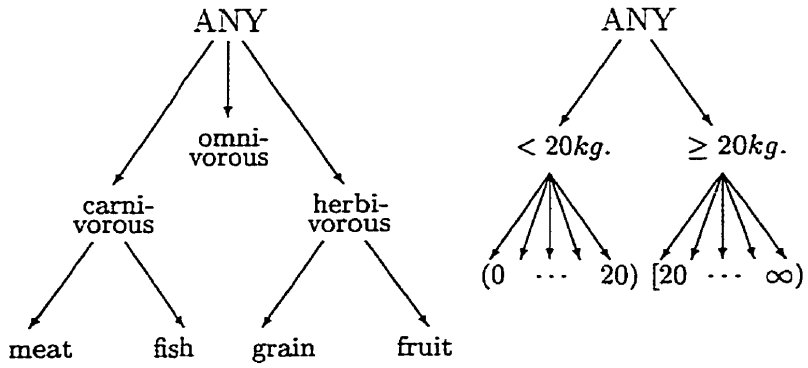
# ↓	Name A_1	Family A_2	Surface A_3	Locomtn. A_4	Diet A_5	Weight(kg) A_6
1	Tiger	felidae	hair	walk	meat	200
2	Cheetah	felidae	hair	walk	meat	50
3	Cayote	canidae	hair	walk	meat	25
4	Chihuahua	canidae	hair	walk	meat	5
5	Giraffe	bovidae	hair	walk	grass	350
6	Zebra	equidae	hair	walk	grass	200
7	Fruit Bat	chiroptera	hair	fly	fruit	2
...	...	...	...	...	...	...
9995	Ostrich	ratites	feather	walk	?	150
9996	Penguin	penguin	feather	swim	fish	10
9997	Albatross	tubenoses	feather	fly	grain	15
9998	Eagle	falcons	feather	fly	meat	15
9999	Macaw	parrots	feather	fly	fruit	5
10000	Anaconda	eunectes	scale	crawl	meat	100

The attribute of a dataset can have several characteristics associated with them. To assist with the discussion, the i^{th} attribute of a dataset will be represented with A_i and the set of possible values for attribute A_i is referred to as $Domain(A_i)$. The Surface attribute in our example is A_3 and its domain is $Domain(A_3) = \{hair, feather, scale\}$. Each attribute A_i may contain either *symbolic* or *numeric* values. The Surface and Diet attributes (A_3, A_5) are examples of symbolic attributes, while the Weight attribute (A_6) is an example of a numeric attribute. The distinction between symbolic and numeric attributes is that only numeric

attributes have a total ordering of its values. A sample ordering for numeric attribute A_i is $(v_{i,1} < v_{i,2} < \cdots\cdots < v_{i,d})$, where $d = |Domain(A_i)|$. A special type of symbolic attribute is the *key attribute*, which uniquely identifies each record in the dataset. Neither the existence of a key attribute, nor the presence of a sorted dataset are required for on-line classification tasks.

Both symbolic and numeric attributes may also have another ordering associated with them in the form of a *concept hierarchy* [33]. Figure 2.1 shows examples of concept hierarchies for both a symbolic and for a numeric attribute. The leaf nodes of the hierarchy contain a single value from the dataset while the root node, at the other extreme, contains every possible value of an attribute, referred to as *ANY*. The internal nodes within the tree contain a set of attribute-values and point to one other node that contains all its members. Even if an explicit concept hierarchy does not exist for an attribute, an implicit one-tier *default* hierarchy always exists in the form of a root node that points directly to all of the attribute's values.

Figure 2.1: Graphical representation of concept hierarchies for the A_5 (*Diet*) and A_{11} (*Weight*) attributes of the sample Animal Kingdom data set. Based on [35].



The *attribute values* within a specific record may also have special characteristics associated with them. Three of these characteristics will be described: *missing values*, *inapplicable values* and *noisy values*. When a particular attribute value is unknown, this missing information can be represented with a question mark (?). In the sample dataset, record 9995 has a missing value for A_5. One reason that this may have occurred, is that this value

became smudged since it was recorded in the field. When a particular attribute value is known not to apply to a particular record, then this is represented with the empty set ($\emptyset$) symbol. If there were a *Hair_Colour* attribute, then the $\emptyset$ symbol could be used for record 10000 to represent that snakes do not have hair. Datasets occasionally do not differentiate between missing and inapplicable values. Finally, noise refers to attribute values that have been incorrectly entered.

2.2.2 Event Vector ($\vec{e}$)

The second input parameter for a classification query is an m dimensional *event vector*: $\vec{e}$. This vector contains the information about the particular instance from the same domain as the dataset, whose class is to be predicted. For example, if a person had observed an animal whose surface was covered in feathers, whose diet appeared to be herbivorous, and whose weight was approximately $9kg.$ then this information would be entered with the following event vector:

Figure 2.2: Sample event vector

?	?	feather	?	herbivorous	$9kg.$

Not all the values of the event vector need to be *instantiated.* As shown in this example, unknown values for a particular attribute are marked with question mark *?* symbol. When an $\vec{e}$ is missing some values, the tasks is referred to as *underspecified.* Possible reasons for underspecified tasks, include that a particular value was too costly to retrieve or because time constraints did not allow for the retrieval of the value. When required, the number of instantiated values for a given $\vec{e}$ are referred to with the cardinality function $|\vec{e}|$. The sample event vector above has cardinality of 3.

2.2.3 Class Attribute Description ($\vec{c}$)

The final input parameter for an on-line classification task is a two dimensional vector $\vec{c}$ that describes the attribute of $\vec{e}$ whose value is to be predicted. The first value within this vector $\vec{c}_1$ references the class attribute, while the second value identifies the level in the hierarchy. To predict an animal's general diet, in our example, (*herbivorous, omnivorous, carnivorous*), then $\vec{c}_1 = 5$ and $\vec{c}_2 = 2$. As a general guideline, the selected class attribute

and hierarchy level should contain a small number of distinct values relative to the number of records in the dataset ($\ll n$). In this way, predictions will not have to be based on small sets of records. Based on this guideline, a key attribute or a numeric attribute with no concept hierarchy, would be poor choices as class attributes. Occasionally, the shorthand A_c will be used to represent the class attribute.

2.3 Output Requirements

The minimal output result required by an on-line classification task is the predicted class of the event. Often however, a high-level rationale for the result may also be required to help a person interpret the validity of the prediction. When this is the case the task is referred to as a *knowledge based* on-line classification task.

2.3.1 Class Prediction Format

A classification result may return the single most likely class or likely classes predicted for an event. If the *Family* attribute is to be predicted, the result may be a single most likely animal such as "Cheetah", or may instead predict the two most likely values "Tiger" OR "Cheetah".

An important companion to the predicted value is a probability measure. For our example above the result could be rewritten to ["Cheetah" 95%]. The use of probability measures along with multi-valued prediction can also help to order the results. The multi-valued example above may now be updated to ["Cheetah" 75% OR "Tiger" 20%]. Because these probabilities only *estimate* the true probability of this particular value, some settings may require that a level of confidence be presented along with each prediction. Currently confidence is commonly expressed with the number of matching records.

2.3.2 Class Prediction Justification

The final requirement for output results is for a justification to be presented in a particular type of representation language. Some of common representation for these functions

are sets of instances [3, 66], rules [53, 65] and decision trees [9, 58]. Each of these three representations is briefly reviewed.

Instance-Based Representation

The simplest representation for the justification of a prediction is a report of records (instances) [6, 59, 66]. The event vector in figure 2.2 may for example return the record 9912 (i.e. *Macaw*) because of its similarity to the event vector. An optional parameter for an instance-based representation request is the number of instances that are to be selected. If this value is k, then the k most similar instance from D will be returned.

Rule Based Representation

A higher-level representation of a prediction's justifications than an instance-based result, is the probabilistic rule. Commonly these rules are constrained to propositional logic and in the form of IF *antecedent* THEN *consequent* [64, 67]. The consequent of this rule represents the prediction either as a single value or a disjunction of values, as described above in Section 2.3.1. The antecedent of these classification rules is commonly a conjunction of propositions (terms), where each proposition represents a condition on a single attribute. For symbolic attributes, each proposition can test against a disjunction of attribute-values and may include negation. Some examples include $(A_i = v_{i,j})$, $(A_i \in \{v_{i,j}, \ldots, v_{i,k}\})$, and $(A_i \neq v_{i,j})$. For numeric attributes, each proposition may contain either a one-sided $(A_i \leq a_{i,j})$ or two-sided $(A_i \in [a_{i,j}, a_{i,k}])$ test, where $a_{i,j} < a_{i,k}$. A sample proposition on the Weight numeric attributes is, $(A_6 \leq 8kg)$ for a one-sided test, and $A_6 \in [0kg, 8kg]$ for a two-sided test.

Decision Tree Representation

The final representation of a prediction's justification, that will be reviewed, is the decision (classification) tree data structure. A decision tree is a hierarchical, sequential classification structure that recursively partitions the instance space into mutually disjoint regions [52]. Decision trees are represented with nodes that are connected by branches. Nodes may be either *internal nodes* or *leaf nodes*. Internal nodes contain a test that creates two or more branches to other nodes. Each internal node or leaf node must be referenced by only one other internal node. One exception to this rule is the *root node* which acts as the entry

point into the structure. Finally, leaf nodes contain the class predictions. If a leaf node presents a probability distribution for all classes instead of just the single best value to be predicted, then the tree is referred to as a *class probability tree* [9].

The types of tests allowed within each node are identical to the propositions described above for the rule based representation. Commonly a distinction is made for decision trees with tests that create only two branches or that test more than one attribute. When nodes have binary branching, the corresponding tree is referred to as a binary tree. Trees whose nodes test against a single attribute are referred to as *univariate* trees; otherwise they are referred to as *multivariate* trees.

To classify $\vec{e}$ with a decision tree, the event vector is passed through the tree structure, starting with the root node and continuing until a leaf node is encountered. The class prediction within this leaf node is associated to the event. Occasionally, a test along this path cannot be performed because $\vec{e}$ is underspecified. When this occurs, all the paths from the node in question must be taken. Once all the relevant leaf nodes have been reached, a special function consolidates the predictions within these leaf nodes into a single prediction.

2.4 Control Requirements

An on-line classification task may place some optional constraints on *how* the classification algorithm may achieve its prediction. Three constraints will be reviewed: method of data access, limits on resources such as disk and memory, and finally limits on the amount of time given for the task to complete.

2.4.1 Data Access Constraints

A constraint that may be posed on a classification algorithm is that it interact directly with a database management system. Historically, classification programs have been developed to interact with in-memory datasets. The first step of these programs is to load a text file version of the dataset into memory. This approach limits the ability to support classifications against the large databases that are being collected [39]. One way to overcome this limitation is to make the classification algorithm more database-aware by fetching each

record from the database as required. This *loosely-coupled* approach however, often encounters poor performance due to the copying of records over a network into the application's address space [2]. A tightly-coupled approach, instead, pushes some of the processing directly to the database management system (DBMS). This approach benefits in part from the extensive research into database query optimization. In the case of a database system with SQL support, the use of the `GROUP BY` operation can help to quickly locate summaries [47]. Already, a SQL Interface Protocol (SIP) is proposed in [40] to assist data mining algorithms with the use of this operator. As more powerful summarization operations, such as the `CUBE` proposal in [31], become available within database query languages, the requirement of direct interaction with a database management system will become more likely.

2.4.2 Space Constraints

The possible constraint on data manipulation showed that it can be important to know the space resources available to a prediction program. Other requirements for space resources include temporary structures that may streamline an algorithm's procedures. While the resource capacities continue to grow, so does the amount of information being stored within these systems. If a classification algorithm has insufficient space to operate, it would be desirable for a space efficient version of the algorithm to take over the classification task.

2.4.3 Classification Interruption

The final constraint on how an on-line classification algorithm achieves its task, is due to support for algorithm interruption. In some situations it is desirable that an interrupted classification algorithm be able to produce a partial result. When a person has interrupted a classification task that has already expended a significant amount of time the person may require that a sub-optimal prediction be returned, rather than no prediction at all. This requirement would be expected in dynamic environments where the value of the prediction rapidly diminishes with time. In these situations, it is better to make use of an algorithm that will make incremental progress towards the classification of the given event.

2.5 Performance Measures

Now that the input, output and control requirements of an on-line classification algorithm have been described, a set of measures is now presented that will rank the performance of these algorithms. The main gauges of a knowledge based on-line classification algorithm's success, are its accuracy, speed (time), resource (space) consumption, and the understandability of the their operation and result [41].

2.5.1 Accuracy

An on-line classification algorithm must be accurate. This gauge of success is commonly stated as the minimization of incorrect classifications, or *error rate* [68]. If on average, algorithm A_1 misclassifies every tenth event based on dataset D, then the algorithm is said to achieve an error rate of 10% on this dataset. If we find an algorithm A_2 that achieves an error rate of 5% for this same dataset, then algorithm A_2 is said to be more accurate than algorithm A_1 on dataset D.

To empirically determine which algorithm is generally more accurate than another algorithm, it is common practice to gather a substantial number of datasets with a variety of characteristics. While, no definitive list of benchmark datasets has been composed, a significant number of commonly tested datasets has developed over time [51].

Aside from general accuracy, another common measure of accuracy focuses on the algorithm's sensitivity to certain "real world" characteristics. The accuracy of algorithm A_2, for example, may quickly degrade when many irrelevant attributes are present within datasets. An understanding of this sensitivity can help with the assignment of a particular algorithm to a particular task. Aside from irrelevant attributes other "real world" characteristics that are known to impair the accuracy of classification algorithms include: *missing attributes*, *underspecified events*, *noise*, *missing values*, and different proportions of numeric to symbolic attributes [9, 68].

2.5.2 Running Time

Even if a classification algorithm is very accurate, it may be unusable if it achieves its result too slowly. The parameters that commonly impact the running time complexity of an on-line classification system are the number of records in the dataset (n), the number of attributes in the dataset (m), the number of instantiated attributes of the event vector ($|\vec{e}|$),

the proportion of numeric to symbolic attributes, and the size of each attribute's domain d. Some algorithm's may take less time for large n while others may be more appropriate for large m or $|\bar{e}|$. To get an understanding for an algorithm's time complexity a theoretical worst-case analysis is commonly reported. [11, 21]. Less common is the use of empirical testing.

2.5.3 Space Usage

As already discussed, an algorithm's space complexity may be crucial to the applicability of the system in certain domains. When a classification request occurs against a very large dataset then it may be crucial that an algorithm have small space requirements. The parameters that commonly impact this measure are: the number of records n, and the number of attributes m. The measure of space complexity commonly excludes the size of the dataset. This assumption helps in the selection of algorithms that act directly against DBMS resident datasets. If a copy of the dataset is created (such as in local memory) it is understood that the space complexity will bounded from below by $\Omega(nm)$.

2.5.4 Understandability

Aside from the three objective measures of accuracy, running time and space usage, some online classification tasks will also measure the understandability of an algorithm's prediction. Some settings may also be interested in the understandability of the algorithm itself [68]. Usually, this performance measure is important in settings were people have to integrate their extensive background knowledge to the problem.

As could be expected, understandability of the predicted results and of the process used to achieve this result is mostly a subjective qualitative measure. However, some guidelines have been developed in both areas. For example an algorithm whose operation is transparent and straightforward is preferable to a complicated *black box* algorithm. An understandable algorithm provides a person with some information with which to judge its suitability to the particular task at hand. This is also true of algorithms that presents a clear rationale for their prediction. Of the reviewed representations low order IF-THEN rules have been found to be generally understandable. Low order instance-based results, such as in case based reasoning, may instead allow the person to work from concrete examples. Finally, within a specific representation language, there may be objective measures of simplicity, such as the

number of proposition within a rule, that can be measured and reported.

2.6 Related Issues

Several general requirements of prediction tasks, while closely related to knowledge based on-line classification, are not within the scope of this framework. Three of these requirements include: regression, batch classification and system guided classification.

2.6.1 Classification versus Regression

As described in Section 2.5, a classification algorithm's accuracy is gauged by the percentage of misclassified instances. This measure of accuracy is certainly appropriate when the prediction is on symbolic values. When the value to be predicted however, is from a numeric attribute, accuracy may instead be measured by the average distance between the predicted value to the true value. Rather than minimizing the error rate, the goal now is to minimize this distance [34]. This measure of accuracy has been well studied in the field of statistical data analysis under the name of regression [20]. The focus of the current framework will be limited to classification because it provides a simpler measure of accuracy that still captures a significant number of prediction tasks. In the future, there may be interest in the study of on-line regression.

2.6.2 On-Line versus Batch Classification

Another separation of classification tasks can be drawn between tasks that require a prediction for a single event, or for a large number of events. Historically, classification research has focused on the latter case. Because of the focus on the class prediction of many events, classification has therefore been generally divided into two phases. The *learning phase* develops a predictive model from a *training set* while the *testing phase* uses this model to quickly predict the class for any given event [9, 58, 48]. This approach will be referred to as *batch classification*. In batch classification tasks, the data set D and class attribute $\vec{c}$ are known well in advance of the classification request for several event vectors. Rather than measuring the amount of time required to classify a single event, batch classification

tasks measure the amount of time required by the learning phase. Once this phase has constructed a classifier, the time required to classify an event based on this structure is assumed to be minimal. Therefore, while a batch-classification algorithm may be applied to on-line classification tasks, its running time performance may not be appropriate for on-line classification tasks. An algorithm that is specifically targeted to on-line classification, has the advantage of not having to develop a classifier that will classify *every* event from the domain.

2.6.3 Data Driven vs. System Guided Classification

The final separation of classification tasks to be reviewed is between those tasks that require guidance about which values to instantiate within the event vector $\vec{e}$ and those tasks, as described in the framework above, that can proceed without this guidance. This separation of tasks assumes that instantiating all the values in the event vector is either not possible or not trivial. If all the values in the event vector are instantiated, then there is no need for guidance. There are situations where the cost and time of acquiring specific data values have to be weighed for the classification task. In the Animal Kingdom example, it may be more time consuming (or costly) to determine the number of teeth that a large animal may have inside its mouth, than it is to determine the number of limbs it possesses, or to estimate the animals mouth shape. On the other hand, even though it is not trivial to get this information, the number of teeth may happen to be a very valuable piece of information to make an accurate prediction. The question now is who determines what values to instantiate. In data driven interaction the agent is assumed to be competent enough to make this determination [64]. In system guided (or active) classification, the algorithm has the ability to suggest which attribute would be the next best value to instantiate [32]. Decision trees structures are an example of a system guided classifier. The root node describes which attribute-value would like produce the most predictive classification. The focus of the framework described in this chapter is for settings that require efficient data driven classification. The reason for this restriction is that investigations into system guided on-line classification may simply require the addition of a separate analysis engine. Therefore, research into a data driven approaches may still be of value to future investigations into interactive classification algorithms.

2.7 Chapter Summary

This chapter presented a framework for knowledge based on-line classification. To facilitate the discussion a simple Animal Kingdom example was first presented. The example was then used to present the input, output and internal control requirements for these tasks. Four performance measures were then described: accuracy, time, space and understandability. Finally, the description of the framework concluded with a review of three areas that are closely related to on-line classification, but out of the scope of this thesis: regression, batch classification and system guided classification.

Chapter 3

Related Work

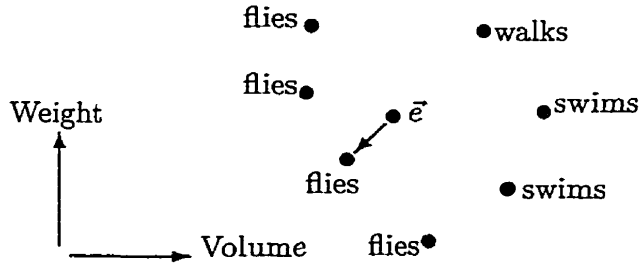
Now that a framework has been proposed for knowledge based on-line classification tasks and algorithms, this chapter reviews several current classification algorithms that may fit into this framework. The survey focuses on machine learning research because of its focus on methods that are compatible with human reasoning. Within machine learning, classification algorithms commonly make use of either a lazy instance-based or an eager model-based approach. An alternate approach that may be more suitable to on-line classification tasks is the use of lazy induction with dynamic relevance analysis. This survey provides an overview of the three approaches. This includes some history of the research, key computation details, and summaries of their general performance results.

The survey will be presented in the following order. Section 3.1 presents an overview of the lazy instance-based approach to classification, with the IB1 algorithm [3] as the representative algorithm. Next, Section 3.2 presents the eager model-based approach to classification, with top-down induction of decision trees [57] as the representative technique. Finally, Section 3.3 presents the more recent approach of lazy induction with dynamic relevance testing. Within this section two techniques are presented: local induction of decision trees and lazy model-based induction. The DBPredictor algorithm presented in this thesis falls into the latter category. This chapter, however, presents the LazyDT algorithm as the representative for lazy model-based induction.

3.1 Instance-Based Learning

One of the simplest methods to predict an event's class, is to recall the past observations that are most similar to this event and to base the prediction on most common class within this group. This approach has been extensively used by instance based learning (IBL) algorithms. In this section the IB1 [6] instance-based learning algorithm is highlighted. This algorithm may be used against datasets with numeric and symbolic attributes, and in the presence of missing attribute-values. Generally IBL algorithms are fast but their accuracy is susceptible to certain "real world" characteristics such as the presence of irrelevant attributes. Finally, while it is easy for a person to understand how the algorithm operates, the low-level representation of the prediction's rationale can be difficult to interpret.

Example 3.1. The following example will show how to predict whether animal $\vec{e}$ can fly via instance-based classification. Assume that $\vec{e}$ is a three dimensional vector which describes the animal's weight, volume and form of locomotion, $\vec{e} = [weight = 0.7, volume = 0.4, locomotion = ?]$. The first step of an instance-based classifier is to evaluate the similarity of $\vec{e}$ and all (n) dataset records. Since all the predicting attributes in this example are numeric, the similarity can be expressed in terms of the geometric distance between $\vec{e}$ and the other records. The figure below presents the relative location of the seven most similar records to $\vec{e}$.



Because both the most similar record to $\vec{e}$ and the larger proportion of the seven most similar instances have class "fly", by the similarity assumption, an IBL algorithm would predict that $\vec{e}$ also flies.

□

One of the earliest applications of the *similarity assumption* to a classification algorithm is found in the *k-nearest neighbour* (k-NN) algorithm proposed by Cover & Hart, 1967 [17].

Since then, the field of machine learning has incorporated this technique within *instance-based learning* (IBL) algorithms. Samples of these algorithms include IB(1 thru 4) [3] and PEBLS [16]. Other related approaches include memory-based learning [66] and case-based reasoning [44].

Given a `Similarity()` function that outputs a numeric-valued similarity, an IBL algorithm will calculate the similarity between an event vector and every record (instance) in the dataset. Based on the similarity values assigned to each record, a `Prediction()` function will return the prediction for the IBL algorithm. The similarity and prediction functions for the IB1 algorithm [6] are presented below. This implementation was used in the empirical studies reported in Chapters 7 and 8.

3.1.1 IB1's Similarity Function

The basis of IB1's similarity function is the inverse of the Euclidean distance between two vectors. This function is shown in Equation 3.1 where the *attribute difference function* $\delta(x_i, y_i)$ is set to $(x_i - y_i)^2$. The similarity between vectors $\vec{x} = [0, 1]$ and $\vec{y} = [1, 0]$, for example, would be $1/\sqrt{(0-1)^2 + (1-0)^2} = \frac{1}{\sqrt{2}} \approx 0.71$. As the two vectors move closer to each other (and the distance between them approaches 0) this similarity measure will return a larger numeric value. This function is undefined when the distance between the two vectors is equal to 0, so the distance is not allowed to become any smaller than some small ϵ.

$$Similarity(\vec{x}, \vec{y}) = \frac{1}{\sqrt{\sum_{i=1}^{n} \delta(\vec{x}_i, \vec{y}_i)}} \tag{3.1}$$

As defined, the current *Similarity*() function is appropriate for datasets that contain numeric attributes with little or no variation in their ranges. Several updates to the attribute difference function $\delta()$ are presented that will allow the IB1 algorithm to be applied against datasets with numeric attributes with large variations in their range and also when attributes are symbolic or contain missing values.

Normalization of Numeric Values

When a dataset's numeric attributes contain large range variations, the current attribute difference function $\delta()$ favours attributes with smaller numerical ranges. To counteract this arbitrary bias, the ranges of numeric attributes are first normalized. Commonly the

normalized range is set to (0,1) with Function 3.2.

$$Normalize\_attribute(x_i, A_i) = \frac{x_i - min(A_i)}{max(A_i) - min(A_i)} \tag{3.2}$$

Example 3.2. This example presents the benefit of numerical attribute normalization. Assume that $\vec{e} = [50, 0.5]$, where the range of A_1 is $[0, 100]$ and the range of A_2 is the much smaller $[0, 1]$. Next, assume the existence of the following two records in the dataset, $\vec{r}_1 = [45, 0.1]$ and $\vec{r}_2 = [60, 0.6]$. Notice that $\vec{e}$ is significantly different than $\vec{r}_1$ on attribute A_2 (i.e. 0.5 vs. 0.1). The raw similarities are presented below:

$$\begin{aligned} Similarity(\vec{e}, \vec{r}_1) &= \frac{1}{\sqrt{(50-45)^2+(0.5-0.1)^2}} = 0.2 \\ Similarity(\vec{e}, \vec{r}_2) &= \frac{1}{\sqrt{(50-60)^2+(0.5-0.6)^2}} = 0.1 \end{aligned}$$

Based on these evaluations, $\vec{e}$ appears to be more similar to $\vec{r}_1$ than to $\vec{r}_2$, $(0.2 > 0.1)$. However, when A_1 is normalized to the range $(0, 1)$, the similarities change to

$$\begin{aligned} Similarity(\vec{e}, \vec{r}_1) &= \frac{1}{\sqrt{(0.50-0.45)^2+(0.5-0.1)^2}} \approx 2.5 \\ Similarity(\vec{e}, \vec{r}_2) &= \frac{1}{\sqrt{(0.50-0.60)^2+(0.5-0.6)^2}} \approx 7.1 \end{aligned}$$

Based on these normalized calculations, $\vec{e}$ is now significantly more similar to $\vec{r}_2$ than to $\vec{r}_1$, as desired. □

Nonnumeric Attributes

The final updates to the attribute difference function $\delta()$ are shown in Function 3.3. These changes to the function allow for datasets that possess either symbolic attributes or missing attribute values. For symbolic attributes, a simple overlap metric is added to the function. When two symbolic values do not match, a difference of 1 is returned, and otherwise, a difference of 0 (identical) is returned. For missing values, the difference will be set to the maximal separation. When one value is not missing, say for example it is equal to 0.2, then the other value is assumed to be 1.0 (for a difference of $.8^2$).

$$\delta(x_i, y_i) = \begin{cases} max(x_i - 0, 1 - x_i)^2 & \text{if } y_i \text{ is missing} \\ max(y_i - 0, 1 - y_i)^2 & \text{if } x_i \text{ is missing} \\ 1 & \text{if both values are missing} \\ (x_i - y_i)^2 & i \text{ is numeric} \\ x_i \neq y_i & \text{symbolic} \end{cases} \tag{3.3}$$

Open Questions

Two open questions remain before this function meets all the requirements of the on-line classification framework.

First, it is unclear how to support attributes with concept hierarchies. One possible solution would be to assign a difference quotient $\in [0, 1]$ to every attribute value and hierarchy node pair. This value would represent a logical difference between each possible combination. For example a diet of "banana" should be presented as more similar to a general diet of *Fruit* than when compared to a diet of "tuna".

$$\delta(A_4 = banana, A_4 \in Fruits) \ll \delta(A_4 = banana, A_4 = tuna) \tag{3.4}$$

These quotient values may be dynamically determined, at some expense, with the value difference metric (VDM) proposed by in [66].

Second, it is unclear how to tightly-couple the updated similarity measure with an SQL-like database interface. No research was located that addresses this question and no solution is clearly apparent. For now, a loosely-coupled solution which copies every record into the application space, may be required.

3.1.2 IB1's Prediction Function

Given the similarity function just described, an instance-based algorithm can calculate the similarity between the event vector $\vec{e}$ and all the records of the dataset. Once this has been accomplished a *prediction function* completes the classification task by basing the prediction on the records that have been found to be most similar. One possible prediction function is to return the class of the 1-*nearest neighbour*, with ties in similarity being resolved randomly. IB1's prediction function however, bases its prediction on the k most similar records (k-*nearest neighbour*). The value for threshold k is commonly optimized for the domain in question.

3.1.3 Performance Characteristics

IBL algorithms perform very well along the measure of running time, but their accuracy suffers under some specific conditions. Specifically while accuracy can be very good for

numeric datasets, it can also degrade for datasets with irrelevant and symbolic attributes. Finally, while the method used by these algorithms is simple to understand, understanding the meaning of its instance-based prediction representation can prove to be difficult against some datasets.

Time and Space Complexity: The time complexity of the IB1 algorithm against an on-line classification task is bounded by $O(nm)$, for datasets with n records and m attributes. Two passes of the dataset are likely. The first determines the ranges of the numeric attributes so that normalization can occur. The second pass then evaluates the similarity of the normalized records to the event vector. If the k most similar records are stored through this second pass, then no other pass is required. In the worst-case, the calculation of the `Similarity()` function requires m computations of the attribute difference function $\delta()$ per record. If we assume a fixed cost of $\delta()$ then the total time complexity is bounded by $O(nm)$. Finally, given that k records will be stored to base a prediction, the space complexity of this algorithm is bounded by $O(km)$.

Accuracy: The low time complexity of IB1 unfortunately comes at the expense of inaccuracy in the face of irrelevant attributes, noisy values and symbolic values. Extensions to IB1-like algorithms, such as IB3, IB4 and PEBLS [3, 16], have been proposed to circumvent these problems. Examples of these extensions include IB3's use of a *probation period* to locate *reliable instances*, IB4's use of *attribute weight settings* to deemphasize irrelevant attributes and PEBLS use of the value distance metric (VDM) to make the attribute difference function $\delta()$ more informative for symbolic attributes. The significant increase in computational complexity of these updates however, make these updated algorithms more applicable to batch classification than to on-line classification tasks.

Understandability: Instance-based algorithms achieve a mixed response on the measure of understandability. The operation of these algorithms is easy to understand but their instance based prediction can be difficult to interpret. A prediction for example, that is based on twenty five dataset records each of which is described by fifty attribute-values may be very difficult to interpret even by a domain expert. If on the other hand, a prediction is based on one to five records, each with ten values, an expert may be able to formulate a reasonable hypothesis about why these particular instances support this particular class

prediction.

3.1.4 Summary

IBL algorithms apply the similarity assumption to classification tasks. Once an event vector's similarity has been evaluated against all records in the dataset the k most relevant records (instances) are returned. This review presented the IB1 algorithm. Its similarity function supports datasets with numeric attributes (including those with wide differences in their ranges), symbolic attributes and missing attribute-values. While the IB1 algorithm achieves a linear running time bounded by $O(nm)$, its accuracy is vulnerable to some specific types of domains. Several methods have been proposed to address this problem, but they significantly increase the algorithm's running time complexity. Finally, this approach has the further advantage of being simple to understand, however in datasets with large numbers of attributes, the instance-based representation (of k records) may not provide a clear justification about a particular class prediction.

3.2 Top-Down Induction of Decision Trees

Another highly developed method of classification is the construction of decision trees by way of top-down induction [9, 58]. Section 3.2 already reviewed the structure of decisions trees, this section will instead review how to create these structures with top-down induction. This will include a brief review of heuristic measures. Finally, the general performance of these algorithms will be compared and contrasted to the performance of instance-based algorithms. Generally, decision tree algorithms are slower than IB1-like instance based algorithms, but this extra time allows the approach to achieve more robust accuracy and produce a higher-level representation.

Top-down induction of decision trees has been extensively researched in the fields of Statistical Data Analysis and Machine Learning, and continues to be actively investigated [52]. One of the first algorithms to construct a decision tree was the Concept Learning System (CLS) by Hunt *et al*, 1966 [37]. The standard references on the top-down induction of decision trees (TDIDT) are Breiman *et al*'s "Classification and Regression Trees" [9] and

Quinlan's description of the C4.5 algorithm in "Programs for Machine Learning" [58]. The former reference is from a statistical data analysis perspective, while the latter presents a machine learning perspective to the problem.

3.2.1 Tree Construction Process

The process of building a decision tree from a data set D is known as tree *induction*. The major challenge to this process is to locate an accurate tree from among the many possible trees. The use of brute force to find a tree that minimizes some measure, such as accuracy, is an NP-complete problem [38]. A data set with just $m = 4$ attributes each of which has $d = 2$ distinct values (i.e. binary) can produce approximately 1.5 billion[1] binary univariate trees. As in other areas of artificial intelligence, TDIDT algorithms work around this problem by investigating a much smaller space of trees and using a heuristic function to locate a "good" solution. Specifically, TDIDT algorithms use a heuristic *evaluation function* to guide a *greedy* tree creation that starts at the root node and proceeds towards the leaf nodes (i.e. *top-down*). While the algorithm could be extended to search along several paths (beams) or with some limited lookahead, these extensions have not shown improved accuracy results [52].

A TDIDT algorithm creates decision trees by adding either internal nodes or leaf nodes to the tree structure starting from the root node. To ensure that TDIDT algorithms terminate, internal nodes must continually divide the dataset into smaller sets of records. This can be accomplished with a constraint that tests be nontrivial and non-redundant [57]. A test such as $A_1 \leq 1.5$, for example would not be allowed if a test for $A_1 \leq 2.0$ has already been placed higher up in the tree. This *divide and conquer* process continues until a *stopping criterion* forces the creation of a leaf node. Two minimal rules are required. The first rule forces the creation of leaf node when no more internal nodes may be added. The other rule creates a leaf node when no more records are available in D to support further branching. The prediction within a leaf node is based on the class distribution of the records in D that reach this node. If most records that reach a leaf node have a class of "Fly", but some records also have a class "Swim", then this information would be reflected in the prediction found within the leaf node.

[1] $\prod_{i=0}^{md} 2^i \binom{md-i}{1}$

Pruning

The process of tree construction just described is particularly suited to environments without missing attribute, noisy entries or irrelevant attributes. If this is not the case, such as with "real-world" datasets the algorithm will often place a leaf node beyond the location of the most accurate leaf node[56]. To counteract the effects of this *overfitting*, two *pruning* techniques have been found to be effective. The *pre-pruning* technique expands the stopping criterion to stop tree growth earlier [57]. One such rule would be to avoid leaf nodes that match too few records [67]. The intuition of this rule is that predictions based on too few records become unreliable. Empirical evidence however, has shown that pre-pruning is not as effective as the *post-pruning* technique [9, 49]. In *reduced-error* post-pruning, the decision tree is generated with the normal stopping criterion but the tree is then tested against a portion of D that was set aside. Internal nodes of the tree are converted into leaf nodes, if an improvement in accuracy is noted on the records that were set aside.

3.2.2 TDIDT Evaluation Functions

The general search technique just described above requires the existence of heuristics (evaluation) functions to guide the generation of the decision trees. Because no backtracking is used by these (greedy) algorithms, there is a strong incentive to choose a good heuristic. As with instance-based algorithms, the evaluation function has been shown to significantly impact the accuracy of the resulting decision tree [50]. In the case of decision tree algorithms, evaluation functions attempt to return tests that most resemble the underlying structure of the domain. Chapter 6 provides more details about evaluation functions that may be used by top-down induction. For now, two categories of evaluation functions are briefly reviewed, those based on information theory and those based on class distribution separation.

Impurity Measures

The group of functions most commonly used to guide top-down induction, are based on information theory [65]. Their strategy is to reduce the randomness or *impurity* over all the nodes of the decision tree. Within this category the best known function is *entropy function*, Function 3.5, proposed by Quinlan[2] [55]. The function assigns an impurity measure to each

[2] where $0 \log_2 0 = 0$

class probability distribution vector. Because the base of *log* is set to 2 the result of this measure is expressed in terms of *bits*.

$$i(\vec{v}) = -\sum_{i=1}^{c} v_i \log_2 v_i \tag{3.5}$$

Based on this measure of a node's impurity, the induction process attempts to select tests that minimizes the impurity of the resulting class distributions.

Class Separation Measures

An alternative to impurity based evaluation functions are functions that quantify the *distance* between class probability distribution vectors. Less attention has been given to this class of evaluation function although several benefits over impurity based functions have been documented [26]. The greater the distance reported between two class probability distribution vectors by distance function $\delta(\vec{\alpha}, \vec{\beta})$, the greater the likelihood that the corresponding test matches the underlying structure of the domain. Two specific measures exist, one based on the angle between class probability vectors, $ORT()$ [25], and the other, based on the Euclidean distance between two class distribution vectors, $DI()$ [67].

3.2.3 Performance Characteristics

Because of the significant differences in the approaches used by TDIDT and IBL algorithms, it is not too surprising that their performance characteristics are also significantly different. The main trade-off occurs between accuracy and time. TDIDT generally achieves more robust accuracy than IBL, but this improvement comes at the expense of increased time complexity. Finally, TDIDT algorithms also return more concise justifications of their predictions, but IBL algorithms are simpler to understand.

Time and Space Complexity: The previous section on IBL algorithms showed that their time complexity is bounded by $O(nm)$ and their space complexity by $O(m)$, for tasks with n records and m attributes. The time complexity for top-down decision tree algorithms, on the other hand, is bounded by $O(nm^2)$ when attributes are symbolic and $O(n^2m^2)$ when attributes are numeric [21]. The increase complexity for numeric domains is due to the sorting required to locate the appropriate split [58]. While there have been several proposals to reduce the complexity of TDIDT algorithms with respect to m and n, these updates have

resulted in a loss of accuracy [11]. Finally, the space complexity of TDIDT algorithms is bounded by $O(nm)$ [57].

Accuracy: The increased time and space complexity of decision tree algorithms is balanced by their increased accuracy. This improvement however is not universal. Domains that particularly benefit from the use of decision trees are those with with many irrelevant attributes and with noisy attribute-values [52].

Understandability: Finally, the understandability of decision tree based results also differ from that of instance-based algorithms. TDIDT algorithms are not as simple to understand, as IBL algorithms, particularly when a complex evaluation function is used. The prediction result of TDIDT algorithms however, can be easier to interpret than an instance based results. The path of the decision tree that is followed by an event to make the prediction, elucidates which attributes contributed in a significant way to the prediction.

3.2.4 Summary

Top down induction of decision trees (TDIDT) is a well understood approach to classification. The survey of this approach focused on the production of binary univariate trees. While locating the most predictive tree is an NP-complete problem, the use of a greedy divide and conquer strategy has been shown to be very effective at approximating the optimal tree. To guide this process several heuristic evaluation functions have been developed. When compared to lazy instance-based algorithms, decision tree algorithms are generally slower but more accurate. The running time is significantly longer in numeric domains while the accuracy is superior in the presence of irrelevant, noisy and symbolic attributes. Finally, the understandability of TDIDT algorithms and their prediction result also differs from IBL algorithms. The tree path based result, allows for simpler interpretation of the reasons for the given prediction, however, the algorithm itself, particularly when used in conjunction with a complex evaluation functions, is not as simple to follow.

3.3 Lazy Algorithms with Dynamic Relevance Testing

When choosing an algorithm for an on-line classification task, IBL algorithms such as IB1 [3] and TDIDT algorithms such as C4.5 [58] provide performance results that are at the opposite extremes of the time and accuracy dimensions. Two recently proposed techniques integrate characteristics of both IBL and TDIDT to achieve a middle ground of performance when applied to on-line classification. Local induction of decision trees [30] is a hybrid of IBL and TDIDT, that first gather a substantial portion of similar instances, and then develops a decision tree based on this subset of records. Lazy model-based induction [29, 46], on the other hand, is an integrated combination of lazy learning and model-based induction that develops only the portion of the model required to classify the event vector. DBPredictor, uses this latter technique. Like IBL algorithms, both techniques focus their effort on the classification of a particular event vector. This characteristic is commonly associated with lazy learning algorithms [4]. Both techniques however, also make use of the dynamic relevance analysis of top-down induction algorithms to determine which attributes are relevant to the task at hand. Dynamic relevance analysis further ensures that an algorithm focuses its effort on the classification task at hand rather than trying to locate the globally predictive attributes to facilitate the classification of all possible events.

While both techniques are reviewed, more coverage will be given to lazy top-down induction. The reason for this bias is due to the latter's generation of a knowledge-based justification to their predictions. Within the review of lazy model-based induction, greater emphasis is given to the LazyDT algorithm because the DBPredictor algorithm is described in significant detail in Chapters 4 thru 6.

3.3.1 Lazy vs. Eager Algorithms

Recall that TDIDT algorithms such as C4.5 require a significant amount of effort to develop a complete classification tree before they can make a prediction. Similarly, IBL algorithms such as IB4, perform a signification amount of processing to counter the shortcomings of the IB1 algorithm. The two proposed techniques of local induction of decision trees and lazy top-down induction, on the other hand, perform little global processing to achieve their classification result. They instead possess the lazy learning characteristic of deferring all processing, until they receive classification requests [5].

Most of the computation expended by TDIDT algorithms is directed at the development

of a complete classification model in the form of a decision tree. In doing so, they are able to determine which attributes are relevant to the classification task, and with the use of pruning can also avoid dependence on unreliable records. This style of classification is referred to as eager learning because it eagerly develops a complete model in anticipation of a large batch of future classification requests [4]. When applied to on-line classification tasks however, most of their computation occurs in portions of the decision tree that are unrelated to the classification task at hand. A purely lazy algorithm such as IB1, on the other hand, carefully stays within the scope of each individual on-line classification task it receives.

Unfortunately, the accuracy of k nearest-neighbour based algorithms, such as IB1, is sensitive to irrelevant attributes and noisy records. To mitigate for these effects, several methods have been proposed to locate both the relevant attributes and unreliable records of a dataset. The current methods however, eagerly develop a global model of relevance and reliability that is appropriate for all future classification requests. In the case of attribute selection (feature selection), current methods require processing in the order of $O(n^2m^2)$ before they can proceed to the classification task at hand [22]. A purely lazy algorithm for on-line classification should instead determine which attributes are relevant and which records are reliable, strictly to the prediction of the given event.

3.3.2 Local Induction of Decision Trees

One way to construct a lazy algorithm that also performs dynamic relevance analysis is to create a hybrid algorithm that makes use of both instance-based learning and decision tree induction. Such an algorithm has been proposed in [30]. While the algorithm was originally targeted at "interactive data mining", its approach fits the framework for on-line classification. Local induction of decision trees has three key steps. First, it retrieves a substantial number of records that are *similar* to the event vector, in $O(nm)$ to $O(n^2m)$ time. Next, the algorithm induces a model (in this case a decision tree) from this set of records in $O(km^2)$ to $O(k^2m^2)$ time, where $k \ll n$. Finally, the algorithm uses the induced model to classify the event vector. In a sense, the algorithm makes use of IBL to constrain its effort on the task at hand and makes use of TDIDT to perform dynamic relevance

analysis. The accuracy of this algorithm appears to be promising. Initial experiments show the algorithm to be superior to decision tree algorithms. The running time of the algorithm varies from a good $O(nm + k^2m^2)$ to a poor $O(n^2m + km^2)$.

Local Neighbourhood Selection

One of the main challenges to local induction of decision trees is to dynamically determine the characteristics of the group of records from which the decision tree is to be based on. Each domain will likely require that a different number of records be passed to the decision tree construction step. Two of the three proposed solutions are summarized. Finally because of its use of an IBL component, it is noted that support for concept hierarchies and tightly-coupled RDBMSs interaction remain open questions.

Two methods of dynamically determining the appropriate number and characteristics of the records passed to the decision tree step are considered. The "local induction voting" algorithm passes k sets of *similar* records to the decision tree step which then proceeds to produce k trees. The size of each set ranges from 1 to k. To determine which class prediction to return, a round of voting occurs between all k trees. The "multilayer composite neighborhood" algorithm, on the other hand, returns a single group of records. The group is identified by selecting records that are not only similar, but are similar in ways that also include other records. This process however requires a worst-case[3] running time of $O(n^2m)$. It is unclear whether this solution supports undiscretized numerical attributes.

Finally, because of its use of an instance based learning component, this lazy induction algorithm shares the open issues of IBL algorithms, discussed in Section 3.1.1, with respect to support for concept hierarchies and tightly-coupled RDBMS support.

Performance

Because this is a new algorithm, its performance is not yet fully understood. However, initial empirical results show that the algorithm achieves superior accuracy when compared to decision tree algorithms. Also, the running time is reported to be "similar" to IBL algorithms. Finally, with respect to understandability, it is unclear what representation is used to support the classification result. Local induction of decision trees will inherit the representation problems of IBL presented in Section 3.1.3.

[3]This can be lowered to $O(mn\log^{m-1} n)$ complexity for narrow (small m) data sets

Summary

Local induction of decision trees is a promising approach to on-line classification. With its combination of IBL and TDIDT algorithms, the algorithm can focus its effort to the classification task at hand while being able to dynamically determine attribute relevance. The current obstacles to its use includes its worst-case running time complexity of $O(n^2m)$. Also, this approach inherits some of the difficulties encountered with IBL algorithms. These include the lack of a knowledge based prediction justification and the inability to interface directly to an RDBMS.

3.3.3 Lazy Model-Based Induction: LazyDT

Another approach to a lazy algorithm that performs dynamic relevance analysis is to continually specialize the portion of the predictive model that applies to the event vector in question. Two algorithms that make use of lazy model-based induction are LazyDT and DBPredictor. The LazyDT algorithm proposed in Friedman *et al*, 1996 [29] continually specializes the portion of a decision tree relevant to the classification of $\vec{e}$. The algorithm in a sense, dynamically creates the path of the decision tree that the event vector would have taken, had an entire decision tree been constructed. The DBPredictor algorithm proposed in Melli, 1996 [46], on the other hand, continually specializes a probabilistic `IF-THEN` rule that classifies the event vector in question. The remainder of this section focuses on the LazyDT algorithm and some of its open areas. The DBPredictor algorithm is presented in the next chapter.

Overview

LazyDT, **Lazy** Decision Tree, is a classification algorithm that returns a class prediction supported by a path of a binary univariate decision tree. While LazyDT is a clear descendant of TDIDT algorithms, it differs from these in several ways. Its most novel contributions are its lazy path generation and its variation of the entropy evaluation function. Also of interest, are its required discretization of numerical attributes, its use of a very conservative specialization step size and its use of one-level lookahead to break ties.

Example 3.3. Given dataset D, 4-dimensional event vector $\vec{e} = [a_{1x}, a_2 = 5, a_3, ?]$ and class attribute description $A_c = A_4$, the LazyDT algorithm proceeds through the following steps. Because LazyDT operates only on discrete symbols, it first discretizes all numerical attributes in D. The next step of the algorithm is to create the root node of the virtual univariate binary decision tree. Several alternative nodes (hypotheses) are generated and then evaluated against a heuristic function. The nodes generated by LazyDT include tests of the form $(A_i \neq v_i)$. Nodes are then tested with a variation of an *entropy()* based evaluation function. The test that is deemed most predictive is selected. Next, another round of hypothesis generation and evaluation is performed. This process continues until a stopping criterion is encountered. When this occurs, LazyDT concludes by returning the tree path just generated and also returns the class distribution of the records in D that also reach the leaf node of this path. Below is a sample result which shows a path of a tree (in brackets) and its leaf node (far right).

$$\Big[(A_1 \neq a_{1x}) \rightarrow (A_2 \neq [2,4]) \rightarrow (A_1 \neq a_{1y})\Big] \Rightarrow (A_4 \in [3,0,0]) \tag{3.6}$$

The result can be interpreted to mean that because $\vec{e}_1$ in not equal to a_{1x} or a_{1y} and because $\vec{e}_2$ is not in the range [2,4], LazyDT predicts a class of $A_4 = c_1$ for event vector $\vec{e}$. The range [2,4] for attribute A_2 would have been generated within the discretization step. □

Discretization

In settings where datasets have numerical attributes, LazyDT's first step is to discretize these. Once the dataset has been discretized, its tree path can contain tests of similar format, regardless of the type of attribute they refer to. Several effective discretization algorithms are available [23]. LazyDT happens to use the algorithm by Fayyad & Irani, 1993 [25]. This algorithm discretizes each numerical attribute independently from each other. With the use of the entropy function, the algorithm recursively places cuts within the attribute, based on their ability to minimize the entropy of the resulting subsets. It proceeds in this fashion until the minimum description length stopping criterion is met [61]. Because discretization sorts the values of each attribute, the running time of this algorithm is bounded by $O(m \times n \log n)$. It is unclear whether discretization can be efficiently performed directly against a relational database and without having to physically change the values of the attributes. In Section

4.8 we propose a method that avoids the need for discretization.

Hypothesis Generation

The constraints imposed by LazyDT on its hypothesis generator is that it develop nodes for a univariate binary decision tree [12] that applies to event $\vec{e}$. To achieve this, each node performs a true or false test against only one attribute. Within this framework, LazyDT chooses to generates tests in the form of $A_i \neq a_{ij}$, where $a_{ij} \in Domain(A_i)$ except for the value that the value being used by the event vector, $\vec{e}_i$. For example, if there are $d = 10$ unique values in the three predicting attributes of Example 3.3, the algorithm will generate and test 27 $(3 \times 10 - 3 \times 1)$ root node hypothesis. The final path must contain fewer than 27 $(m(d-1))$ nodes. Support for hypothesis generation against attributes with concept hierarchies has not been proposed. In Section 4.8, we propose a method that supports concept hierarchies.

Lookahead

Early investigations of LazyDT's performance discovered that many ties occurred between the possible paths. This may be due to its conservative hypothesis generation. Rather than greedily selecting a random node from among the winners, LazyDT performs a one-step lookahead, on all nodes that achieve an information gain value that is within 90% of the highest value. If some nodes still tie, a random selection is performed from these. No results are reported on the impact on accuracy of this enhancement.

Evaluation Function

To determine which of the several possible paths to commit to, LazyDT makes use of the entropy measure. Unlike the approach used by TDIDT algorithms to calculate the information gain along both branches of a test, LazyDT's evaluation function measures the change in entropy only along the path that the event vector will follow. Because a comparison of this approach has not been performed, Chapter 6 explicitly presents both types of evaluation function calculations in greater detail. We refer to standard test as sibling-sibling and LazyDT's as parent-child. Chapter 7 then reports empirical results which indicate that the parent-child variation of the evaluation function degrades, rather than improves, a classifier's accuracy.

Overspecialization

LazyDT stops developing the path of the tree when either no more information can be inferred from the domain (i.e. lack of dataset records or lack of information about $\vec{e}$) or when all the records in the dataset that reach the current node are all of the same class. As with TDIDT algorithms, this approach will likely lead to trees that overfit datasets which contain real-world characteristics such as irrelevant attribute, noise values and missing attributes. Section 3.2.1 reviewed the common use of pruning to overcome this problem. LazyDT's lack of a method to mitigate against overfitting is noted in [29]. In Section 6.5 we propose a simple pruning method to mitigate against overfitting and in Chapter 7 we report that this approach remedies DBPredictor's vulnerability to overfitting.

Performance

Because LazyDT is a new algorithm, its performance is not yet fully understood. Initial empirical results however, show that the algorithm may be more accurate than decision tree algorithms. This accuracy is achieved at the expense of greater computational complexity than TDIDT algorithms. Finally, its path based result provides informative justification of the prediction.

Accuracy: The initial research into LazyDT's accuracy shows that it may be more accurate than C4.5r5. When tested on 28 datasets, the algorithm achieved a lower error rate than C4.5r5 on 16 (57%) of these datasets. In Chapter 7, we show that this accuracy will likely extend to domains with irrelevant attributes and underspecified event vectors. We also show the relative performance to the IB1 algorithm.

Time and Space Complexity: The time required by LazyDT to classify an event vector in a domain that does not need discretization is stated to be bounded by $O(nmd)$, where d is the largest number of unique values in any given attribute. This appears to be incorrect, however, no detailed analysis is available in [29] to formally test this claim. Briefly however, we saw in Example 3.3, that the path of the result may contain up to md nodes. Further, $md - i$ hypothesis will be generated and tested at node i, assuming that no

lookahead is performed. Therefore, the maximum number of hypothesis that can be tested is $< m^2d^2$. When the time complexity of discretization is added the time complexity of LazyDT appears to be bounded by $O(mn \log n + nm^2d^2)$. The analysis of DBPredictor in the Sections 4.12 and 5.4 will help to substantiate this informal claim. Since decision tree algorithms that also perform an initial discretization step achieve a running time complexity of $O(mn \log n + nm^2)$, LazyDT appears to require a longer running time than TDIDT algorithms. Finally, no space complexity or timing comparisons are provided. Section 5.4 indicates that LazyDT's space complexity is likely bounded by $O(nmd)$.

Understandability LazyDT returns a classification that is supported by a path through a univariate binary decision tree. This result can allow a person to understand the rationale for the prediction, and make some informal decisions about the prediction's soundness. Because the internal nodes test for inequality, the path appears to be interpreted by negative information. For example, animal $\vec{e}$ may be predicted to be mammal because it does not have feathers and was not born from an egg. It is unclear whether a more positive statement, such as animal $\vec{e}$ likely being mammal because it drank milk as a child, may be more effective. Finally, no information is available on the length and make-up of the generated paths to determine how concise or unwieldy they may be.

Summary

Lazy top-down induction is a promising approach to knowledge base on-line classification tasks. With its lazy version of model based specialization, this approach focuses its effort to the classification task at hand and dynamically performs attribute relevance analysis. The LazyDT and DBPredictor algorithms make use of this approach. This review described some of the open questions of this approach that are investigated within this thesis. The model which the LazyDT algorithm specializes is a binary univariate decision tree. Initial empirical results support the claim that the algorithm achieves greater accuracy than the C4.5r5 decision tree algorithm. Due to its conservative hypothesis generation and one-level lookahead, the time complexity of the algorithm is greater than for decision trees. Future investigation will likely determine whether its conservative approach to hypothesis generation, and its use of a lookahead step are key to the algorithms increased accuracy. Finally, as with decision trees, the tree path result is generally informative.

3.3.4 Summary of Lazy Induction with Dynamic Relevance Analysis

Instance-based learning (IBL) and top-down induction of decision trees (TDIDT) algorithms, result in dramatically different performance behaviours for on-line classification tasks. TDIDT is generally more accurate while IBL is faster. Two recent techniques reach a compromise to this performance dichotomy with the use of lazy induction and dynamic relevance analysis. Local decision trees and lazy model-based induction, both focus their energies to returning an answer strictly for the task at hand. This includes the effort expended at relevance analysis. Local induction of decision trees achieves this compromise with a hybrid approach that first uses an IBL component, and then passes the results to a TDIDT component. Lazy model-based induction, on the other hand, only develops the portion of the model that is appropriate for the task at hand. Of this latter technique, only the LazyDT algorithm is reviewed. The presentation of the DBPredictor algorithm is left for the next chapter.

3.4 Chapter Summary

This chapter reviewed several approaches that are applicable to knowledge based on-line classification tasks. Instance-based learning (IBL) algorithms were found to be very fast. However their accuracy performed poorly in the face of irrelevant attributes unless a significant amount of processing is added. Top-down induction of decision trees (TDIDT) on the other hand takes a significant amount of time but does achieve a more robust level of accuracy than (IBL). Finally, two recent approaches were reviewed under the category of lazy induction with dynamic relevance analysis. A brief comparison between lazy and eager algorithms was presented to assist with the presentation of local induction of decision trees and the lazy top-down induction technique used by LazyDT and DBPredictor.

Chapter 4

DBPredictor Algorithm

The next three chapters propose and analyze a lazy model-based classification algorithm named DBPredictor, that is targeted at knowledge based on-line classification tasks. This chapter presents the core of the search technique used by the algorithm. The next chapter describes an alternate version of the algorithm's search technique that will return the same result, but achieves a faster running time at the expense of greater space requirements. The alternate version of the algorithm is referred to as the time efficient version and is labeled with a subscripted "T" (DBPredictor$_T$). The third chapter, describes several versions of the heuristic function, that may be used by either search technique.

A high-level presentation of the DBPredictor algorithm has been previously described in Melli, 1996 [46]. The presentation within this chapter provides greater detail and analysis, and also introduces three enhancements:

1. A dynamic numerical proposition specialization method that avoids the use of global discretization.
2. Support for tightly-coupled integration with an SQL database.
3. Support for attributes with concept hierarchies.

This chapter presents DBPredictor's space efficient search technique in the following order. Section 4.1 gives an overview of the algorithm by way of example. Section 4.2 reviews the input parameter requirements, while Section 4.3 reviews the rule based representation of the prediction result. Next, Section 4.4 introduces the high-level call to the DBPredictor algorithm, while Sections 4.6-4.11 present the algorithm's supporting procedures. To show

the possibility of a tightly-coupled implementation to a database, Section 4.5 describes an SQL-based interface for DBPredictor. To conclude the chapter, Section 4.12 presents an analysis of the algorithm's running time and space complexity.

4.1 Overview

The DBPredictor algorithm produces a probabilistic IF-THEN rule that classifies a specific event. To achieve this result, the algorithm requires information about the event and a dataset of records from the same domain as the event The algorithm begins by generating a very general rule that covers all the records in the dataset and then proceeds to incrementally specialize this rule in ways that are expected to increase the rule's predictive value. To facilitate the presentation of DBPredictor's detailed operation, a simple example is now presented that proceeds through the algorithm's main phases.

Example 4.1. This simple example of the DBPredictor algorithm assumes the presence of symbolic attributes and the following classification request:

- Dataset D with $n = 100$ records and $m = 4$ symbolic attributes.
- Unlabeled event $\vec{e} = [a_1, a_2, a_3, ?]$ with 3 symbolic predicting values[1]
- Class attribute $A_c = A_4$ with c=2 unique class values $\{c_1, c_2\}$
- Class attribute distribution $[c_1 = 30, c_2 = 70]$ in dataset D.

DBPredictor first generates a very general *seed rule* (r_0) based on the overall distribution of the class attribute.

$$r_0 : \quad (A_{\{1,2,3\}} = a_{any}) \quad \rightarrow A_4 \in [30, 70]$$

If applied to $\vec{e}$ this rule predicts that the event will likely be of class $A_4 = c_2$ with 70% ($\frac{70}{30+70}$) probability.

Because DBPredictor possesses some information about $\vec{e}$, the algorithm will generate several rules that are slightly more specialized than rule r_0. Assume that three rules are

[1]The fourth attribute is set to unknown because this is the attribute whose value is to be predicted.

generated at this specialization step:

$$\begin{aligned} r_1: &\quad (A_1 = a_1) &&\rightarrow A_4 \in [16, 64] \\ r_2: &\quad (A_2 = a_2) &&\rightarrow A_4 \in [14, 6] \\ r_3: &\quad (A_3 = a_3) &&\rightarrow A_4 \in [30, 70] \end{aligned}$$

Note that as required, all three rules continue to apply to $\vec{e}$. The second rule (r_2) can be interpreted to read that 20 (14+6) records in the dataset have $A_2 = a_2$. Of these 20 records, 14 belong to class c_1, and the remaining 6 belong to class c_2.

Next, each of these three rules is tested with a heuristic function to determine which specialization has the most predictive value. Assume that the second rule is selected.

DBPredictor now generates another set of rules that are somewhat more specialized than r_2 while still applying to event $\vec{e}$. Assume the two following rules are generated:

$$\begin{aligned} r_4: &\quad (A_2 = a_2) \wedge (A_1 = a_1) \rightarrow &&A_4 \in [0, 0] \\ r_5: &\quad (A_2 = a_2) \wedge (A_3 = a_3) \rightarrow &&A_4 \in [14, 6] \end{aligned}$$

Unfortunately, these two rules are of no predictive value. Rule r_4 did not match any records in the dataset, and rule r_5 has the same distribution as r_2. Therefore, the search stops and DBPredictor returns rule.

$$\begin{gathered} \texttt{IF } A_2 = a_2 \\ \texttt{THEN } A_4 = c_1(70\%) \texttt{ OR } A_4 = c_2(30\%) \\ \text{(support: 20 records)} \end{gathered}$$

From this result, a person would likely predict class c_1 for the given event. □

4.2 Input Parameters

This section summarizes the input parameter framework already presented in the Section 2.2 and highlights any requirements that cannot be met by DBPredictor. DBPredictor requires three input parameters $(D, \vec{e}, \vec{c})$. D represents the dataset that will be used to base the prediction, $\vec{c}$ represents the symbolic attribute of D whose value is to be predicted and $\vec{e}$ contains the information about the specific event vector that the classification is requested for. The main shortfall from the framework is in how missing values within dataset records are handled.

4.2.1 Dataset D

The first input parameter is a dataset D with n records and m attributes. The algorithm meets all the requirements for datasets specified in the framework except for natural support of missing attribute values. Attributes may contain symbolic and numerical values, and may be also described by a concept hierarchy.

Two methods exist within DBPredictor to handle missing values in a dataset. The algorithm may either treat these values as distinct from all other values or may set the missing value to be equal to the value of the event vector under consideration. As an example of the second approach, if the event vector's value for attribute A_i were set to v_i, then all the records in the dataset with missing values on this attribute would have this value logically set to v_i. The specific method used by the algorithm is determined with the use of an optional parameter. The default behaviour, however, is to treat missing values as distinct values. The main reason for this default behaviour is the doubling of time complexity required to test whether a record's value is missing or not regardless of the percentage of missing attributes in the dataset.

4.2.2 Event Vector $\vec{e}$

The second input parameter required by DBPredictor is an m-dimensional event vector $\vec{e}$. This parameter contains the information about the event that is to be classified in the form of attribute-value pairs. The vector maps directly to the m attributes in dataset D. The sample event vector presented in Example 4.1 ($\vec{e} = [v_1, v_2, v_3, ?]$) can be reinterpreted to $[A_1 = v_1, A_2 = v_2, A_3 = v_3, A_4 = ?]$.

4.2.3 Class Attribute $\vec{c}$

The final input parameter is a two dimensional $\vec{c}$ that represents the class attribute whose value is to be predicted. The first value of the vector is the identifier of the class attribute itself. If the value of the fourth attribute is to be predicted, then $\vec{c}_1 = 4$. The second dimension of the vector contains the level within the concept hierarchy for this attribute that is to be predicted. If for example, this value is set to 1 ($\vec{c}_2 = 1$) the prediction would occur among the nodes at the first level down from the root of the concept hierarchy. If no concept hierarchy exists on the class attribute this portion of the parameter is unused.

4.3 Output Format

Given the three input parameters described above $(D, \vec{e}, \vec{c})$, DBPredictor outputs its class prediction result as a probabilistic IF *antecedent* THEN *consequent* classification rule. The objective of this representation is to provide a practical and understandable mechanism to explicitly represent the prediction [64, 69]. Since the rule's consequent contains the rule's prediction, it will be described first, and will be immediately followed with a description of the rule's antecedent.

4.3.1 Rule Consequent

DBPredictor's class prediction is contained in the consequent of its classification rule result. Instead of containing just the most likely class value prediction, the rule consequent contains the probabilistic distribution of the most likely classes. This is similar to the representation used by probabilistic decision trees [9]. The use of class probability distributions over single best class prediction will be of assistance when only weak predictions can be made about the potential two or three classes that are most likely to occur. The result from Example 4.1

$$r_2: \quad antecedent \quad \rightarrow A_4 \in [c_1 = 14, c_2 = 6]$$

predicts that most likely classification is likely class c_1 but may occasionally turn out to be class c_2.

4.3.2 Rule Antecedent

The rationale for DBPredictor's class prediction is contained within the antecedent of its classification rule result. The rule antecedent is structured in conjunctive normal form (CNF) with up to $m-1$ ANDed *propositions*. Each proposition is a true or false membership test on a single attribute. P_i will refer to the proposition that performs a test on the i^{th} attribute in the dataset, A_i.

The form of the membership test for each proposition varies on the type of attribute it references: symbolic, numeric or hierarchical. For symbolic attributes, the test will be an equality test against the single value from the attribute's domain. For example a proposition against a "Colour" attribute could take the form of $(A_{colour} = \text{"}red\text{"})$. Numeric attributes, on the other hand, are supported with the use of a membership test within a two-sided

numeric interval. The test that a value for the A_{weight} must be greater than or equal to -1.5 and less than or equal to 4.5 can be represented with $(A_{weight} \epsilon [-1.5, 4.5])$. For hierarchically structured attributes, the test is for membership for any node in the hierarchy, such as $(A_5 \in Carnivorous)$ where $Carnivorous = ("meat", "fish")$.

A special *null proposition* on attribute i that is always True is represented with the notation $P_i = (A_i = ANY)$, where ANY represents the set of all unique values within attribute A_i. Based on the null proposition, the *null rule* is true for every record and event vector from that domain. A sample antecedent with propositions against symbolic, numerical, hierarchical attributes and a null proposition is presented below:

$$r: \quad (A_1 = a_1) \wedge (A_2 \epsilon [a_{2min}, a_{2max}]) \wedge (A_3 \epsilon \{a_{3a}, a_{3b}\}) \wedge (A_4 = ANY) \quad \rightarrow consequent$$

To conclude the discussion on rule antecedents it will be shown that a given event vector will likely be covered by a large number of antecedents.

Proposition 4.1. A fully instantiated m dimensional event vector $\vec{e}$ can be covered by at least $2^{(m-1)}$ logically distinct antecedents.

Proof. First map each element $\vec{e}_i$ to proposition P_i that tests for equality to this value $(A_i = \vec{e}_i)$. Because a fully instantiated m-dimensional event vector has $m-1$ predicting elements, the set of all these propositions, S, has a cardinality of $m-1$ $(|S| = m-1)$. Next, we create the set of all subsets of S (its *powerset*). Each of these subsets maps to a logically distinct rule antecedent that covers $\vec{e}$. Because a power set has been proven to have a cardinality of $2^{|S|}$ [8], the number of logically distinct antecedents that will cover an m-dimensional event vector $\vec{e}$ is $\geq 2^{m-1}$, as desired. □

Example 4.2. An event vector with three instantiated values $\vec{e} = \{v_1, v_2, v_3\}$ has at least eight rules that will cover it $(2^{|\{v_1, v_2, v_3, ?\}|} = 2^3 = 8)$. If $P_i \leftarrow (A_i = a_i)$ these rules include: $\{\{\emptyset\}, \{P_1\}, \{P_2\}, \{P_3\}, \{P_1 \wedge P_2\}, \{P_1 \wedge P_3\}, \{P_2 \wedge P_3\}, \{P_1 \wedge P_2 \wedge P_3\}\}$, where $\{\emptyset\}$ represents the *null rule*. If the propositions are expanded beyond equality (=) tests, such as for numeric attributes, the number of antecedents that can cover a rule would be larger. □

4.4 DBPredictor() Algorithm

DBPredictor's underlying search strategy is to perform a greedy top-down search through the space of candidate rules. Based on its input parameters, DBPredictor first retrieves a starting rule with the seed_rule() procedure. This rule is usually a very general rule that covers the event vector. Next, the top_down_search() procedure is initiated with the seed rule. Finally, the rule returned by the search is reported. A pseudo-code version of this algorithm is presented in Algorithm 4.1.

Algorithm 4.1 DBPredictor() pseudo-code

Input: training database D, event vector $\vec{e}$ and class attribute description $\vec{c}$.
Output: rule whose antecedent covers $\vec{e}$ and whose consequent predicts the value of A_c at the concept hierarchy level specified in $\vec{c}_2$.
Method:
1: $r \Leftarrow$ seed_rule$(D, \vec{e}, \vec{c})$
2: predicted_$r \Leftarrow$ top_down_search$(r,(D, \vec{e}, \vec{c}))$
3: return(predicted_r)

The time and space complexity of DBPredictor is equivalent to the complexity of its two procedures, along with the data they exchange. The seed_rule() procedure is described in Section 4.6 and the top_down_search() procedure is described in Section 4.7.

4.5 P_SIP() Procedure

Before we proceed with the description of the seed_rule() procedure, and any other procedure that requires information from the dataset, the P_SIP() procedure is presented because it provides the interface to the dataset. The P_SIP() procedure implements the "pure SIP" procedure of the SQL Interface Protocol (SIP) proposed in [40]. In this way, we present how DBPredictor may be tightly-coupled to a dataset supported by a relational database. As discussed in Section 2.4.1, the tight-coupled approach has been recently shown to be significantly superior to a loosely-couple approach that manipulates each record individually [2]. Also, in the near future, this procedure may be updated to use the more efficient CUBE operator defined in [31].

Given rule antecedent $r_{antecedent}$, database D and class attribute description $\vec{c}$, a call to procedure P_SIP($r_{antecedent}$, D, $\vec{c}$) returns the distribution of the values in attribute A_c, for the records in D that are covered by rule antecedent $r_{antecedent}$. In [40] this is achieved with the following SQL statement:

SELECT $\phi, f(*)$ FROM D
WHERE ψ GROUP BY ϕ

For DBPredictor, this statement is transformed by the P_SIP() procedure into the following SQL statement:

SELECT A_c, COUNT(*) FROM D
WHERE $r_{antecedent}$ GROUP BY A_c
where A_c represents the dataset column at the hierarchy level specified in $\vec{c}_2$.

An example of the transformation is provided in Example 4.3.

Example 4.3. Assume that the P_SIP() procedure receives the following arguments:

1. $r_{antecedent} = (A_1 = a_1) \wedge (A_2 \epsilon [a_{2min}, a_{2max}]) \wedge (A_3 \epsilon \{a_{3a}, a_{3b}, a_{3c}\}) \wedge (A_4 \in ANY)$
2. $|D| = n = 100$ records
3. $A_c = A_5$, with two class values (c_1, c_2).

The rule antecedent in this example contains three propositions, each against a symbolic, numeric and hierarchical attributes. Null propositions are discarded. The method that each proposition is transformed is detailed below:

1. For symbolic attributes a single membership test is required. In this example the following test would be performed $(v_1 = a_1)$.
2. For numeric attributes two tests need to be performed to determine if the record's value is within the range specified in the proposition. In this example the record is tested against $\Big((v_2 \leq a_{2max})$ AND $(v_2 \geq a_{2min})\Big)$.

3. When an antecedent's proposition tests an internal node of a concept hierarchy then the record's value is tested against the members within this node. In this example the following test would be performed $\Big((v_3 = a_{3a})$ OR $(v_3 = a_{3b})$ OR $(v_3 = a_{3c})\Big)$.

The resulting SQL statement for this example is:

$$\begin{aligned}
\texttt{SELECT} \quad & A_4, \texttt{COUNT(*)} \\
\texttt{FROM} \quad & D \\
\texttt{WHERE} \quad & \Big((v_1 = a_1)\Big) \\
\texttt{AND} \quad & \Big((v_2 \leq a_{2maxq}) \texttt{ AND } (v_2 \geq a_{2min})\Big) \\
\texttt{AND} \quad & \Big((v_3 = a_{3a}) \texttt{ OR } (v_3 = a_{3b}) \texttt{ OR } (v_3 = a_{3c})\Big) \\
\texttt{GROUP BY} \quad & A_4
\end{aligned}$$

A possible result of this SQL statement is: $A_4 \in [c_1 = 30, c_2 = 70]$. This shows that 100 records were covered by $r_{antecedent}$, and of these, 30 of them had value c_1 in attribute A_4 and the remaining 70, had value c_2.

□

In the example above, each of the 100 records may have been tested against the 3 propositions in the rule's antecedent, for a maximum of 300 proposition tests. This statement is formalized below:

Proposition 4.2. Given a rule antecedent with i propositions and a dataset with n records, the P_SIP() procedure visits all n records and performs $\leq n \times i$ proposition tests

Proof. To determine whether a record is covered by the antecedent of rule r', all of its i propositions need to be tested against the record. Because there are n records in the dataset, there will be at most $n \times i$ tests for any given call to the P_SIP() procedure, as desired. □

4.6 seed_rule()

The first task of DBPredictor is to locate the seed rule that will be used to initiate the top-down search. A very general rule is required. The `seed_rule()` procedure described in Procedure 4.2 simply returns the *null rule* by calling the `P_SIP()` procedure to summarize the overall distribution of the dataset's class attribute.

Procedure 4.2 `seed_rule()`

Input: $(D, \vec{e}, \vec{c})$: dataset D, event vector $\vec{e}$, and class attribute description $\vec{c}$.
Output: The *null rule* for class attribute $\vec{c}$.
Method:
1: $r \Leftarrow \emptyset$ {initialize the rule}
2: $r_{consequent} \Leftarrow$ P_SIP$(\emptyset, D, \vec{c})$ {retrieve the distribution of $\vec{c}$}
3: return r

Proposition 4.3. Given a dataset with n records and class attribute with c values, the worst-case time and space complexity for the `seed_rule()` procedure just described are $O(n)$ and $O(c)$ respectively.

Proof. From proposition Proposition 4.2, all n records in the dataset are visited, therefore the procedure's time complexity is bounded by $O(n)$. Next, because the procedures requires a data structure to contains a value for each of the c unique class values, the space complexity is bounded by $O(d)$. □

Discussion: The $\vec{e}$ parameter is not used in the acquisition of the null rule. In the future, however, it is envisaged that information about the event vector may be used to improve the seed rule result.

4.7 top_down_search()

The bulk of DBPredictor's effort occurs within the top_down_search() procedure. The procedure can be said to perform a greedy top-down search through a constrained rule space. Procedure 4.3 presents a pseudo-code description of the procedure. The top_down_search() procedure makes use of four sub-procedures. First, the generate_antecedents() procedure determines the set of candidate rules that are to be tested within each specialization step. It achieves this by returning a set of skeleton rules that have only their antecedent portion defined. This procedure is further described in Section 4.8. Next, for each of the rule skeletons, their consequent is populated by get_consequent(). This procedure interfaces with the database to determine which records are covered by a particular rule. The procedure is described further in Section 4.9. Next, each rule is evaluated by the F() heuristic. This function is briefly reviewed in Section 4.10, but is more thoroughly presented in Chapter 6. Once all the rules have been generated and tested, the highest-valued rule is located with the best_rule() procedure[2]. This procedures is described further in 4.11. The recursion stops when no more rules can be generated, or when all tested rules do not achieve a predictive improvement over the parent rule.

Procedure 4.3 top_down_search() pseudo-code

Input: (r, P): r is the current rule and P contains the algorithm parameters $(D, \vec{e}, \vec{c})$.
Output: A rule that cannot be specialized further
Method:

```
1: R ⇐ generate_antecedents(r, P)
2: for all rule r' ∈ R do
3:    r'_consequent ⇐ get_consequent(r', r, P)
4:    r'_value ⇐ F(r', r)
5: end for
6:
7: best_r' ⇐ best_rule(R)
8: if (best_r' ≠ ∅) then
9:    return(top_down_search(best_r', P))
10: else
11:   return(r)
12: end if
```

[2]Ties are randomly resolved

4.8 generate_antecedents()

One of the main steps in DBPredictor's top-down search procedure is to generate the next set of rules to be investigated. Proposition 4.1 showed that number of valid antecedents may very large. To constrain the search space, the generate_antecedents() returns up to $m-1$ rules each time that it is called. The procedure enforces this constraint by specializing each one of the $m-1$ propositions in the parent rule. The way that each proposition is specialized depends on the attribute type that it references: symbolic, numeric or hierarchical.

Before describing the method used by the generate_antecedents() procedure to generate its set of rules, a re-cap is provided on the requirements that these rules must meet. First, all rules must conform to the representation presented in Section 4.3. This includes, for example, the constraint that each proposition P_i refer only to attribute A_i. The second constraint on rules is that they must cover the given event vector. This is required so that the algorithm's prediction continues to apply to the original request. Finally, all rules must be more specialized than the current rule. This requirement ensures that the top-down search makes continuous progress. This also guarantees algorithm termination because rules will logically cover a smaller of dataset records upon each specialization.

4.8.1 Proposition Specialization

A high-level presentation of how each specialization occurs is initially presented for each individual attribute type. Particular attention is given to the specialization of numeric attributes. This is followed by a Procedure 4.4 which describes the steps taken by generate_antecedents(in pseudo-code.

The general approach used in proposition specialization is to make use of a hierarchy on the attribute. When attributes do not have an explicit hierarchy defined over them, an implicit hierarchy is, in a sense, dynamically generated. With the presence of a hierarchy, a proposition's test can be specialized by simply testing against a more specific node in the hierarchy. In this way a more specialized proposition is generated and only one new rule is proposed per proposition.

Hierarchical Attributes: The existence of an explicit concept hierarchy requires that the proposition on this attribute be updated to point to the next node down the hierarchy towards the value contained in the event vector, $\vec{e}_i$. If a proposition is currently testing against *Diet=herbivores*, it is simple to change the proposition to test against the more specific *Diet=fruits*, if the diet of the animal in question is $\vec{e}_{Diet} = bananas$.

The first specialization attempt on proposition P_i $(A_i \in \{ANY\})$ will result in proposition $(A_i \in \{N_{1i}\})$, where N_{1i} represents the internal node in the concept hierarchy on the first level of the hierarchy and in the direction of $\vec{e}_i$. Further specialization can continue up to and including the leaf node of the hierarchy. Given an attribute with an h-level hierarchy, a proposition on this attribute may be specialized up to h times. Calls for specialization on a proposition that tests on a leaf node are forwarded to either the specialization of numeric or symbolic attributes. Commonly the leafs of hierarchies on symbolic attributes already refer to the record level data so no further specialization would be possible.

Symbolic Attributes: When symbolic attributes do not possess an explicit concept hierarchy a very simplistic implicit hierarchy is used instead. This implicit hierarchy is made up of a root that includes all possible values of this attribute (ANY) and the leaf nodes include the d_i individual symbolic values on this attribute. The first time a specialization is attempted on a proposition that refers to a symbolic attribute A_i, the proposition is simply updated from $(A_i = ANY)$ to $(A_i = \vec{e}_i)$. A proposition on a symbolic attribute can therefore be specialized only one time. In Example 4.1, rule r_2 was generated by specializing the proposition $(A_2 = ANY)$ in rule r_0 to $(A_2 = a_2)$. After this no further specialization could be performed on P_2.

Numerical Attributes: Unfortunately, the method described above for symbolic attributes would result in too significant a specialization if applied to numerical attributes. If, for example, $\vec{e}_2 = 6.5$ and the range on attribute A_2 is $min = 0.5$ and $max = 9.0$, then generating the proposition $(A_2 = 6.5)$ would likely result in a rule that covers few, if any, database records. One approach around this problem is to discretize all numerical attributes. Performing discretization directly against a relational database however, would be time consuming. It may also require that a copy of the database be created to avoid a negative impact on the other applications that also use the database.

To circumvent this problem, propositions on numerical attributes are set to perform a

two sided test ($A_i \in [\vec{e}_i - \delta, \vec{e}_i + \delta]$), where $\delta > 0$. The proposition in our example above may now be set to ($A_2 \in [6.5 - \delta, 6.5 + \delta]$). The question now is how to determine the appropriate δ for each proposition specialization request? DBPredictor achieves this by making the new δ' somewhat smaller than the δ used by its parent's proposition. The proportion of this shrinkage is defined with an internally set fraction named `num_ratio` (numerical partitioning `ratio`).

$$\begin{aligned} \delta' &\Leftarrow \frac{\delta}{\texttt{num\_ratio}} \\ P_i' &\Leftarrow \left(A_i \in [\vec{e}_i - \delta', \vec{e}_i + \delta']\right) \end{aligned}$$

A preprocessing step is required to determine the δ for the *seed rule* that covers all the records in the dataset. After a simple scan through the dataset to locate the *min*, *max* range for each attribute, the δ for each proposition in the seed rule is set to the larger of $(max - \vec{e}_i)$ and $(\vec{e}_i - min)$. In our example above, proposition P_2 in r_0 is set to ($A_2 \in [6.5 - 6, 6.5 + 6]$).

If `num_ratio`=1.5, then the first specialization on this proposition would set δ' to $\frac{6}{1.5} = 4$. A default value for the `num_ratio` internal parameter will be identified during the empirical investigations reported in Chapter 7.

4.8.2 Computational Constraints

Now that the method used by the `generate_antecedents()` has been described, some analysis of its computational constraints is performed. Within this section the analysis is constrained to attributes with 1-level hierarchies, such as symbolic attributes. Later, in Section 4.12.3, the impact of attributes with general h-level hierarchies is investigated.

Lemma 4.1. Given an event vector of symbolic values, the number of propositions in the parent rule will increase by 1 after each call to the `top_down_search()` procedure.

Proof. As defined, the `generate_antecedents()` procedure specializes the parent rule on only one proposition. By definition, only one successful specialization can occur on attributes with 1-level hierarchies. Therefore, each specialization step in symbolic domains must increase the number of propositions in the parent rule by 1, as desired. □

Procedure 4.4 `generate_antecedents()` pseudo-code

Input: $(r, \vec{e}, \vec{c})$: r is the current rule, $\vec{e}$ is the event vector and $\vec{c}$ is the class attribute description. the remaining parameters are

Output: A set of rules that are more specialized than r on or by only one proposition.

Method:

1: $R \Leftarrow \{\emptyset\}$
2: **for all** attributes i that are instantiated in $\vec{e}$ **do**
3: $P_i \Leftarrow$ rule r's current proposition on attribute i
4: $r'.a \Leftarrow r.a - P_i$ {antecedent of rule r' does not contain P_i}
5: $P_i' \Leftarrow \{\emptyset\}$ {new proposition on attribute i}
6:
7: **if** A_i is "Hierarchical" and P_i does not test on a leaf node **then**
8: $N \Leftarrow$ current node in hierarchy tested by P_i
9: $N' \Leftarrow$ descend from N down hierarchy towards $\vec{e}_i$
10: $P_i' \Leftarrow (A_i \in N')$
11:
12: **else if** A_i is "Symbolic" and P_i tests against ANY **then**
13: $P_i' \Leftarrow (A_i \in \{\vec{e}_i\})$
14:
15: **else if** A_i is "Numerical" **then**
16: $min \Leftarrow min(P_i)$ {minimum value currently tested for in P_i}
17: $max \Leftarrow max(P_i)$ {maximum value currently tested for in P_i}
18: $\delta \Leftarrow \frac{(max-min)/2}{\texttt{num\_part}}$
19: $P_i' \Leftarrow (A_i \in [\vec{e}_i - \delta, \vec{e}_i + \delta])$
20: **end if**
21:
22: {update R one if a valid new rule was identified}
23: **if** $P_i' \neq \{\emptyset\}$ **then**
24: $r'.a \leftarrow r'a \wedge P_i'$
25: $R \Leftarrow R \bigcup r'$
26: **end if**
27: **end for**
28: return R

Example 4.4. Example 4.1 demonstrates the constraint of Lemma 4.1. In the example, the `top_down_procedure()` procedure was called against the $[(A_2 = a_2() \rightarrow A_4]$ parent rule. The `generate_antecedents()` procedure then proceeded to generate two candidate rule antecedents $[(A_2 = a_2) \wedge (A_1 = a_1)]$, and $[(A_2 = a_2) \wedge (A_3 = a_3)]$. Both candidate rules have one more proposition than the parent rule. Regardless of which rule is chosen, Lemma 4.1 will hold. □

Corollary 4.1. Given an event vector of symbolic values, there are $i - 1$ propositions in the parent rule during the i^{th} call to `top_down_search()`.

Proof. By induction on i, if $i = 1$, the procedure is called with the null rule which indeed has 0 propositions.

Assume that the result is true for arbitrary i. We want to prove it for $(i + 1)$. Consider the rule with k propositions from the i^{th} call to the procedure. By the induction hypothesis, $k = i - 1$. By Lemma 4.1 each specialization step adds one proposition, so that the number of propositions at specialization step $i + 1$ is at most $k + 1$. This is equal to $(i + 1) - 1$, as desired. □

Corollary 4.2. Given an m dimensional event vector of symbolic values, the number of calls to the `top_down_search()` procedure is $\leq m$.

Proof. By Corollary 4.1 the parent rule at the i^{th} specialization step has $i - 1$ propositions. At the m^{th} specialization, the parent rule has $m - 1$ propositions. By Lemma 4.1 further specialization would require an additional predicting attribute. Therefore the specialization stops on the m^{th} specialization, as desired. □

Example 4.5. Example 4.1 demonstrates the constraints of Corollary 4.1 and Corollary 4.2. In summary, the i^{th} call to the `top_down_search()` procedure occurred on a rule with $i - 1$ propositions. Furthermore, the event vector with three $(m - 1 = 3)$ predicting symbolic attributes $(\vec{e} = [a_1, a_2, a_3, ?])$ required four calls to the `top_down_search()` procedure. The initial call to the procedure was against the null rule with 0 propositions. The three recursive calls $(i = 2, 3, 4)$ occurred against the following parent rules: $[(A_2 = a_2) \rightarrow A_4], [(A_2 = a_2) \wedge (A_1 = a_1) \rightarrow A_4]$, and $[(A_2 = a_2) \wedge (A_1 = a_1) \wedge (A_3 = a_3) \rightarrow A_4]$. □

The recent analysis focused on the computational behaviour of the `top_down_search()` procedure based on the knowledge of how the `generate_antecedents()` procedure specializes rules. The next step of analysis presents the number of rules that are likely to be generated by the `generate_antecedents()` procedure at any given specialization step.

Lemma 4.2. Given an m-dimensional event vector of symbolic values and a parent rule that contains j propositions, the `generate_antecedents()` procedure will create $\leq (m-1)-j$ sibling rule antecedents.

Proof. By induction on j, if $j = 0$ (null rule), the procedure generates a rule with one proposition for each of the $m-1$ predicting attributes, so the total number of rules is indeed $\leq (m-1)-0 = m-1$.

Assume that the result is true for a parent rule with arbitrary j propositions. We want to prove it for $(j+1)$. Consider the k possible sibling rules generated from a parent rule with j propositions. By the induction hypothesis, $k \leq (m-1)-j$. By Lemma 4.1 when the number of propositions is increased by 1 $(j+1)$ there is one less proposition to specialize on. Therefore the number of siblings is $k-1$. This is $\leq ((m-1)-j)-1 = (m-1)-(j+1)$, as desired. □

Example 4.6. Example 4.1 demonstrates the constraint of Lemma 4.2. In one particular instance from the example, the second call of the `generate_antecedents()` procedure is called against the $[(A_2 = a_2() \rightarrow A_4]$ parent rule. Because there are three predicting attributes $(m-1=3)$ and because the parent rule has one proposition, two ((4-1)-1) rule antecedents were generated: $[(A_2 = a_2) \wedge (A_1 = a_1)]$, and $[(A_2 = a_2) \wedge (A_3 = a_3)]$ □

Summary

This section demonstrated some basic constraints on DBPredictor's rule search for symbolic domains. This information will be used in Section 4.12 to analyze DBPredictor's running time complexity.

4.9 get_consequent()

Once a set of rule antecedents has been generated, the get_consequent() procedure is called to construct each rule's consequent. For a given rule's antecedent ($r'_{antecedent}$), the procedure returns a summary of the values for the class attribute (A_c) for all the records in dataset D that are covered by this antecedent[3]. To show how this task can be achieved in a tightly-coupled way, the procedure, presented in Procedure 4.5, will simply make use of the P_SIP() procedure. Recall that P_SIP() achieves its task by testing every record in the dataset against every proposition in r'.

Procedure 4.5 P_SIP based get_consequent()

Input: $(r', r, D, \vec{c}\,)$: r' is the rule under investigation, r is the parent rule, D is the dataset and $\vec{c}$ is the class attribute description.
Output: The consequent of rule r' based on the records that r' covers in dataset D.
Method:
1: $r'_{consequent} \Leftarrow$ P_SIP($r'_{antecedent}, D, \vec{c}\,$)
2: return r'

4.10 F() Heuristic Function

Once a candidate rule had been generated within a specialization step, its predictiveness may be estimated with the heuristic function, F(). Since Chapter 6 is dedicated to the investigation of possible heuristic functions for DBPredictor, for now we assume the existence of a function F(r',r) that returns a numeric value for a rule under investigation (r') and its parent rule (r). The F(r', r) function is used by DBPredictor to estimate the predictive value of specializing from parent rule r to sibling rule r'. A rule r'_i is taken to be more predictive than rule r'_j if F(r'_i, r) > F(r'_j, r).

[3]Note that the procedure also receives the rule's parent, r. This parameter however, is reserved for use by the time efficient version of the procedure described in Section 5.2.

4.11 best_rule() Sub-Procedure

Once all the candidate rules have been generated and tested within a specialization step, the best_rule() procedure is called to select the parent rule for the next specialization step. The current version of the procedure greedily selects the rule that achieved the highest heuristic value greater than zero. When a tie occurs, a random selection from among the tied rules is performed.

As described, this procedure requires minimal computation. In future versions, a more refined selection criteria may be performed from among the "top" candidate rules. In LazyDT [29], for example, a one-level lookahead is performed on all nodes that have attained heuristic values within 90% of highest ranked node. Because current research has not found lookahead to be generally beneficial [52], the best_rule() will continue to act greedily.

4.12 Complexity Analysis

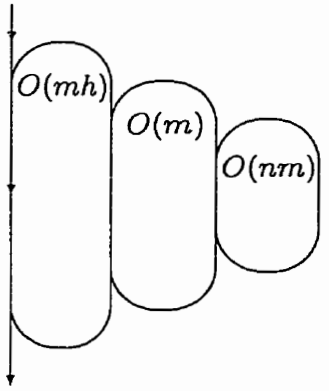
Now that the description of the DBPredictor has been completed, this section analyzes its worst-case running time and space complexity. This analysis is based in large part on the computational constraints of the individual procedures already presented in previous lemmas and propositions. In summary, the algorithm is shown to have, in the presence of symbolic attributes, a running time complexity bounded by $O(nm^3)$ and a space complexity bounded by $O(m^2)$, where n is the number of records and m is the number of attributes. Finally, in the presence of numerical attributes or concept hierarchies, the running time complexity is shown to be linearly dependent on the depth h of the longest explicit or implicit hierarchy.

4.12.1 Running Time Complexity

As Algorithm 4.1 shows the time complexity of DBPredictor is composed of the the seed_rule() and top_down_search() procedures. Proposition 4.3 demonstrated, the seed_rule() procedure is bounded by $O(n)$. For the top_down_search() procedure three Lemmas have been presented for symbolic domains. Corollary 4.2 demonstrated that the number of recursive calls to the top_down_search() procedure is bounded by m. Lemma 4.2 demonstrated that during each of these calls the number of rules generated, is bounded by $(m-1)-i$, where i is the number of propositions in the parent rule. Finally, Proposition 4.2 demonstrated that the number of proposition tests is bounded by the $n \times i$. Figure 4.1 presents a high-level

synthesis of these results.

Figure 4.1: Graphical representation of the $O(nm^3h)$ running time complexity for the space efficient version of DBPredictor, where n is the number of records, m is the number of attributes, and h is the height of the longest concept hierarchy on these attributes. The tallest loop represents the $\leq mh$ number of specialization steps of the `top_down_search()` procedure. The second loop represents the $\leq m$ number of rules generated by the `generate_antecedents()` procedure at each specialization step. The third loop represents how the `get_consequents()` procedure tests each of the up to n dataset records against the $\leq m$ propositions within each rule.



Theorem 4.1. Given a fully instantiated m-dimensional event vector of symbolic attributes and a dataset with n records, the `top_down_search()` procedure will test $\leq \frac{nm(m+1)(m+2)}{6}$ propositions.

Proof. :

Lemma

1. By Corollary 4.2, the `top_down_procedure()` is recursively called up to m times.
2. By Corollary 4.1, the i^{th} call to the `top_down_procedure()` is on a parent rule with $i-1$ propositions.
3. By Lemma 4.2, the `generate_antecedents()` procedure returns $m-i$ sibling rules for a parent rule with $i-1$ propositions.

4. By Proposition 4.2, each call to the get_consequent() procedure, requires $n \times i$ proposition tests for each rule with i propositions.

This is summarized by: $\sum_{i=1}^{m} (m-i)ni$

$$\sum_{i=1}^{m}(m-i)ni = n\left(\sum_{i=1}^{m} mi - \sum_{i=1}^{m} i^2\right) \tag{4.1}$$

$$= n\left(m\frac{(m)(m+1)}{2} - \frac{(m)(m+1)(2m+1)}{6}\right) \tag{4.2}$$

$$= n\left(\frac{m \times 3}{2 \times 3} - \frac{(2m+1)}{6}\right)m(m+1) \tag{4.3}$$

$$= \frac{n(m-1)m(m+1)}{6} \tag{4.4}$$

□

Example 4.7. Example 4.1 demonstrates the constraint of Theorem 4.1. In summary, the example involves four ($m = 4$) attributes, a dataset with n=100 records, and event vector $\vec{e} = [a_1, a_2, a_3, ?]$. Thus we would expect $\frac{100(4-1)(4)(4+1)}{6} = 1,000$ proposition tests. In fact, six distinct rules with a cumulative total of 10 propositions were generated and tested. The table below summarize the number of proposition tests required to determine the cover of each rule:

#	Antecedent	Propositions	Tests
1	P_1	1	1(100)
2	P_2	1	1(100)
3	P_3	1	1(100)
4	$P_2 \wedge P_1$	2	2(100)
5	$P_2 \wedge P_3$	2	2(100)
6	$P_2 \wedge P_1 \wedge P_3$	3	3(100)
SUM		10	1,000

□

Corollary 4.3. In symbolic domains, DBPredictor's running time complexity is bounded by $O(nm^3)$

Proof. By Theorem 4.1, DBPredictor tests up to $n(m-1)m(m+1)/6$ propositions. Since $n(m-1)m(m+1)/6 \leq cnm^3$, for some constant $c > 0$, the worst case running time complexity of DBPredictor is bounded by $O(nm^3)$, as desired. □

Discussion

Each of the $\leq n(m-1)m(m+1)/6$ rules require that their predictive value be estimated by the F() heuristic function. Beyond some constant overhead, this calculation is bounded by the width of the class probability distribution vectors (c). If there are two classes (e.g. true/false) the calculation of functions will require a set of calculations for each of the two classes. The complexity of calculating the evaluation function however, is commonly omitted in classification research and is therefore omitted from any further reporting here as well.

Finally, in database environments it may be of interest to separate the number of database calls from the running time complexity. Informally, DBPredictor, makes a number of database calls bounded by $O(m^2)$, and each of these calls involves a query of $O(nm)$ complexity.

4.12.2 Space Complexity

Assuming that the space taken up by the dataset is not included, DBPredictor's space complexity is briefly analyzed to be bounded by $O(m^2)$. The main data structures created within each specialization step contain the information about the generated sibling rules. Once the highest valued rule is discovered the other rules, including the parent rule, are discarded. By Corollary 4.1, we know that the step with the largest number of sibling rules is the first step, with $m-1$ rules. Assuming that each of these rules allocates space for each of the at most m possible propositions, the space complexity for this version of the algorithm is bounded by $O(m^2)$.

4.12.3 Running Time Complexity with h-level Hierarchies

The computational analysis to date has assumed the presence of only 1-level hierarchy attributes, such as symbolic attributes. When general h-level concept hierarchies are used the complexity increases linearly with the maximum depth h of the explicit or implicit hierarchies contained in the dataset. In the case of numerical attributes, h has been noted to be $\ll \log_{\texttt{num\_ratio}}(N)$.

Lemma 4.1 demonstrated that for attributes with 1-level hierarchies, each specialization step increased the number of propositions in the parent rule by 1. When attributes have h-level hierarchies, h specialization steps are now required to increase the number of propositions in the parent rule by 1. Because of this, instead of n calls, it will now require nh calls to the `top_down_search()` procedure, before all the propositions in the parent rule can no longer be specialized. Therefore, the running time complexity of the algorithm is now bounded by $O(nm^3h)$. Finally, since the number of rules that are generated within each specialization step is unchanged, the algorithm's space complexity remains bounded by $O(m^2)$ in the presence of attributes with h-level hierarchies.

4.13 Discussion

The search algorithm described in this chapter contains three enhancements over previous proposals. One enhancement is the approach to numerical attributes, implemented within the `generate_antecedents()` procedure. This technique frees the algorithm from the space and effort required to discretize a dataset with numerical attributes. The algorithm may now also be used directly against a relational database. Another possible advantage to this approach is that the cuts generated for numeric attributes are customized to the event in hand. This may produce a more accurate result. The validation of this technique is left to experimental study of accuracy in Chapter 7.

The two other enhancements proposed in this chapter are the tightly-coupled SQL support and concept hierarchy support. These two updates also help to expand the ability of the algorithm to operate against more domains. These two approaches proved difficult to validate. First, few classification algorithm currently have tightly-coupled implementations.

Second, few benchmark datasets contain attributes with concept hierarchies. Finally, few classification algorithms currently support concept hierarchies.

4.14 Chapter Summary

This chapter presented the core of the search algorithm used by the DBPredictor algorithm to locate a classification rule. After a brief example of the algorithm's operation, the algorithm's input requirements and `IF-THEN` rule based representation were reviewed. DBPredictor performs a greedy top-down search through the space of candidate rules. The algorithm first composes a high-level a seed rule and then calls the top-down search procedure. This procedure generates and tests several candidate rules and recursively calls itself on the greedily selected rule. The heuristic function F() determines the predictive value of each rule. This function, is described in more detail in Chapter 6. The algorithm's presentation concluded with a demonstration of the algorithm's $O(nm^3h)$ running time complexity, and $O(m^2)$ space complexity, for tasks with n records, m attributes and h-level hierarchy attributes.

Chapter 5

Time Efficient Search Algorithm

This chapter presents an alternate version of DBPredictor's search technique that achieves a lower time complexity than the search technique proposed in Chapter 4, but it does this at the expense of a higher space complexity. Because of this compromise this chapter is said to present the time efficient version of DBPredictor (DBPredictor$_T$), while the algorithm presented in the previous chapter is referred to as the space efficient version (DBPredictor$_S$). The search technique presented in this chapter is more in-line with the current data mining and machine learning algorithms that assume unfettered access to a memory-based dataset array. The time efficient version of the algorithm was developed for two reasons. It allowed for faster testing of DBPredictor's accuracy, and it allowed for a fairer empirical running time comparisons between DBPredictor and the C4.5 and IB1 algorithms.

To achieve its lower running time complexity, DBPredictor$_T$ maintains a list of the records that are covered by each rule. The existence of this list however, minimizes the running time complexity in two ways. The number of records that are tested decrease after each specialization. Only one proposition test is required to determine whether a record is covered by a rule. As will be shown in a latter section, this update increases the algorithm's space complexity from $O(m^2)$ to $O(m^2+nm)$, but lowers the running time complexity from $O(nm^3h)$ to $O(nm^2h)$.

For DBPredictor$_T$ to create, use and release the new list records, three procedures require updates. They are presented within this chapter. Section 5.1 presents the updated `seed_rule()` procedure. Section 5.2 presents the update `get_consequents()` procedure. Section 5.3 presents the updated `top_rule()` procedure. All other procedures remain as stated in the previous chapter. Section 5.4 concludes the chapter with a brief complexity

analysis of the new search technique.

5.1 Updated seed_rule() Procedure

The first procedure that requires modification to support the time efficient version of the DBPredictor algorithm is the seed_rule() procedure. Recall that this procedure returns the consequent of the *null rule* by summarizing the overall distribution of the dataset's class attribute. The updated procedure attaches the list of records that are covered by the rule. Because the null rule covers all the records in the dataset, the list simply points to all the records in the dataset. Procedure 5.1 modifies a non-SIP implementation of Procedure 4.2, with the addition of line item 5.

Procedure 5.1 seed_rule$_T$() pseudo-code

Input: $(D, \vec{e}, \vec{c})$: training database D, event vector $\vec{e}$, and class attribute description $\vec{c}$.
Output: The null rule for class attribute A_c (including rule cover)
Method:
1: $r \Leftarrow \emptyset$ {initialize the rule}
2: **for all** $record_i \in D$ **do**
3: $class \Leftarrow record_i[A_c]$ {set to the class value of this record}
4: $r_{consequent}[class] \Leftarrow r_{consequent}[class] + 1$
5: $r_{cover} \Leftarrow r_{cover} \bigcup (pointer) record_i$ {append the pointer to this record}
6: **end for**
7: return r

The addition of line (5.) should increases the procedure's effort by a constant factor and increase the space requirements by the number of records in the dataset. If the dataset were to contain $n = 100$ records, the *null rule* will now also contain a list of 100 record pointers. The running time complexity of the procedure remains bounded by $O(n)$ while the space complexity increases from $O(c)$ to $O(n + c)$, where c is the number of classes.

5.2 Updated get_consequent() Procedure

The second procedure that requires modification to support the time efficient search is the get_consequent() procedure. In fact, all the saved effort occurs within this procedure. Recall that a call to get_consequent$(r', r, D, \vec{c}\,)$ creates the consequent for rule r' based on the records it covers from dataset D. The space efficient version of the procedure (Section

4.9) achieved this task by testing **every** record in the dataset against **every** proposition in the rule's antecedent. Because a list is now kept of all the records that are covered by the parent rule r, the new procedure can exploit the fact that a sibling rule's cover is a subset of its parents. The procedure can now be optimized in two ways. First, the procedure only needs to test against the records that are covered by the parent rule rather than against every record in the dataset. Second, since the parent and sibling rules differ on only one proposition, the procedure only needs to test against this one proposition to determine if a record that is covered by the parent rule, is also covered by the sibling rule. The updated algorithm of the procedure, labeled with a suffixed $_T$ (get_consequent$_T$()), is described in Procedure 5.2. An example and analysis of the procedure is also presented.

The procedure is no longer presented in SQL format. This procedure may still be tightly-coupled with a relational database, however, this would require the ability to create temporary tables to hold the list of records covered by each rule and would also require that the dataset have an attribute that uniquely identifies each records (i.e. a key attribute).

Procedure 5.2 get_consequents$_T$() pseudo-code

Input: $(r', r, D, \vec{c})$: r' is the rule under investigation, r is the parent rule , D is the dataset and $\vec{c}$ is the class attribute description.
Output: Rule r' with an updated consequent and cover.
Method:

1: $P_i \Leftarrow r'_{antecedent} - r_{antecedent}$ {identify the changed proposition}
2: **for all** $record_i \in r_{cover}$ **do**
3: {test the record against the changed proposition}
4: if P_i is true for $record_i$ **then**
5: {update the consequent based on the class of this record}
6: $class \Leftarrow record_i[A_c]$
7: $r'_{consequent}[class] \Leftarrow r'_{consequent}[class] + 1$
8:
9: {update this rule's cover to point to this record}
10: $r'_{cover} \Leftarrow r'_{cover} \bigcup (pointer)record_i$
11: **end if**
12: **end for**
13: return r'

Example 5.1. Assume that the get_consequent$_T$() procedure was called with the following information:

1. $r' \Leftarrow (A_1 = a_1) \wedge (A_2 \epsilon [a_{2min}, a_{2max}]) \wedge (A_3 \epsilon \{a_{3a}, a_{3b}, a_{3c}\}) \rightarrow A_4$

2. $r \Leftarrow (A_1 = a_1) \wedge (A_3 \epsilon \{a_{3a}, a_{3b}, a_{3c}\}) \rightarrow A_4$.

3. $r.cover = 100$ records

4. $n = 1000$ records

The procedure commences by isolating the proposition that has changed in the sibling rule and assigning it to P_i. In this example the changed proposition P_i is $r' - r = P_2 = (A_2 \epsilon [a_{2min}, a_{2max}])$. With this information, the procedure tests each of the 100 records covered by the parent rule r. Assume the first such record to be tested is $record_1 = D\left[r_{cover}[1]\right] = [v_1, v_2, v_3, v_4]$. This record is tested against proposition P_2. In this example the test would be $[(v_2 \leq a_{2max})$ AND $(v_2 \geq a_{2min})?]$. If this test succeeds then the consequent and cover of the sibling rule is updated accordingly. The remaining 99 records in the cover of r will proceed through this process.

A total of 100×1 proposition tests would be performed by this procedure. This value is significantly than the up to 3,000 ($\leq 1000 \times 3$) proposition tests that would be performed by the space efficient version of the `get_consequents()` procedure. □

By Proposition 4.2, the former `get_consequent()` procedure performed up to m proposition tests for each record in the parent rule's cover. The updated procedure performs only one proposition test for each record. Formally this may be expressed as:

Proposition 5.1. Given a parent rule r with a cover of n' records, the `get_consequent`$_T$`()` procedure performs n' proposition tests.

Proof. There is only one loop in the procedure that cycles through each of the n' records in the cover of r. Because only one proposition test is performed per record the procedures performs $n' \times 1$ proposition tests, as desired. □

5.3 Updated best_rule() Procedure

The final procedure that is updated to complete the definition of the time efficient version of the DBPredictor algorithm, is the best_rule() procedure. Recall that this procedure greedily decides which of the candidate rules will become the next parent rule. During this procedure, an opportunity exists to free most of the space taken up by the list of rule covers. Once the new parent rule has been selected, the space taken up by all the other rules may be freed. The following analysis describes the space complexity within each specialization step.

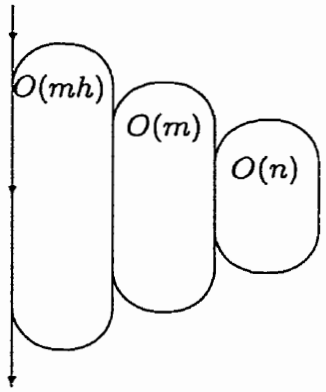
Lemma 5.1. Given an m-dimensional symbolic event vector and dataset with n records, the top_down_search$_T$() procedure requires space for up to $\leq n(m-i)$ record pointers on its i^{th} call.

Proof. By Corollary 4.1, the parent rule of the i^{th} call to the top_down_search() procedure contains $i-1$ propositions. By Lemma 4.2, when the parent rule contains $i-1$ propositions the generate_antecedents() procedure creates $\leq (m-i)$ rule antecedents. Therefore, on the i^{th} call there will be $(m-i)$ rule antecedents. Since each rule may contain a cover of $\leq n$ record pointers, the i^{th} step requires space for $\leq n(m-i)$ record pointers, as desired. □

5.4 Complexity Analysis

This section analyzes the running time and space complexity of the time efficient DBPredictor algorithm proposed above. This analysis is based in large part on the complexity analysis of the individual procedures already presented in previous lemmas and propositions. In summary, the algorithm is shown to have a running time complexity of $O(nm^2h)$ and a space complexity of $O(nm+m^2)$, for tasks with n records, m attributes and attributes with h-level hierarchies. Figure 5.1 presents a graphical representation of the algorithm's running time complexity.

Figure 5.1: Graphical representation of the $O(nm^2h)$ running time complexity for the time efficient version of DBPredictor, where n is the number of records, m is the number of attributes, and h is the height of the longest concept hierarchy on these attributes. The tallest loop represents the $\leq mh$ recursive calls to the `top_down_search()` procedure. The second loop represents the $\leq m$ number of rules generated by the `generate_antecedents()` procedure at each specialization step. The third loop represents how the updated `get_consequents`$_T$`()` procedure tests each of the up to n dataset records against the one changed proposition in the new rule.



5.4.1 Running Time Complexity

To discover the updated algorithm's running time complexity, a theorem is offered on the new number of proposition tests for the algorithm. Based on this result, the upper bound of the algorithm's running time complexity is presented.

Theorem 5.1. Given a fully instantiated m-dimensional event vector of symbolic attributes and a dataset with n records the updated `top_down_search()` procedure will test $\leq nm^2$ propositions.

Proof. :

1. By Corollary 4.2, the `top_down_procedure()` is recursively called up to m times.
2. By Corollary 4.1, the i^{th} call to the `top_down_procedure()` is on a parent rule with $i-1$ propositions.

3. By Lemma 4.2, the `generate_antecedents()` procedure returns $m - i$ sibling rules for a parent rule with $i - 1$ propositions.

4. By Proposition 5.1, each call to the updated `get_consequent`$_T$() procedure requires $n - i$ proposition tests for each rule with i propositions.

This is summarized by: $\sum_{i=1}^{m} (m-i)(n-i)$

$$= \sum_{i=1}^{m} (m-i)(n-i) \tag{5.1}$$

$$= \left(n\sum_{i=1}^{m} m - n\sum_{i=1}^{m} i - m\sum_{i=1}^{m} i + \sum_{i=1}^{m} i^2 \right) \tag{5.2}$$

$$= \left(nm^2 - n\frac{(m)(m+1)}{2} - m\frac{(m)(m+1)}{2} + \frac{(m)(m+1)(2m+1)}{6} \right) \tag{5.3}$$

$$= \left(n\frac{2m^2 - (m^2+m)}{2} + m(m+1)\frac{(2m+1) - 3m}{6} \right) \tag{5.4}$$

$$= \left(n\frac{(m^2-m)}{2} + m(m+1)\frac{(1-m)}{6} \right) \tag{5.5}$$

$$= \left(n\frac{(m^2-m)}{2} + m\frac{(1-m^2)}{6} \right) \tag{5.6}$$

$$= \left(n\frac{(m^2-m)}{2} - \frac{(m^3-m)}{6} \right) \tag{5.7}$$

Because the positive portion of the result, $n\frac{m^2-m}{2}$ is less than nm^2, the `top_down_search()` search procedure will test $\leq nm^2$ propositions, as desired. □

Corollary 5.1. In symbolic domains, the running time complexity of DBPredictor$_T$ is bounded by $O(nm^2)$

Proof. By Theorem 5.1, DBPredictor tests fewer than nm^2 propositions. Since $nm^2 \leq cnm^2$, for some constant $c > 0$, the worst case running time complexity of DBPredictor$_T$ is bounded by $O(nm^2)$, as desired. □

As with the presentation in Section 4.12.3 of the running time complexity with h-level hierarchies for the DBPredictor$_S$ algorithm, when attributes have h-level hierarchies, h specialization steps are now be required to increase the number of propositions in the parent rule by 1. Therefore, the running time complexity of the algorithm is now bounded by $O(nm^2h)$.

5.4.2 Space Complexity

The space complexity for this version of the algorithm is briefly analyzed to be bounded by $O(nm + m^2)$.

Theorem 5.2. Given an m-dimensional symbolic event vector and dataset with n records, the largest number of record pointers required within a call to the `top_down_search()` procedure is $\leq n(m-1)$.

Proof. The smallest value for i is 1 (the call with the null rule). By Lemma 5.1 the i^{th} specialization step generates $(m-1)$ rules each of which may have covers of size $\leq n$. This would require storing $\leq n(m-1)$ record pointers, as desired. □

Due to Theorem 5.2, the space complexity of the time efficient version of the algorithm, is partially bounded by $O(nm)$. Because each of the up to m rules at each specialization step continue to allocate space for the $\leq m$ propositions, the upper bound on the space required by DBPredictor$_T$ changes from $O(m^2)$ to $O(nm + m^2)$.

5.5 Chapter Summary

This chapter presented an alternate version to the search algorithm proposed in the previous chapter. The version within this chapter, DBPredictor$_T$, reduces the algorithm's time

complexity by keeping the list of records that each rule covers. This approach reduces the algorithms time complexity from $O(nm^3h)$ to $O(nm^2h)$, but increases the space complexity from $O(m^2)$ to $O(nm + m^2)$.

Chapter 6

Heuristic Functions

DBPredictor performs a greedy top-down search through a constrained rule space, to locate a rule that predicts the class of event $\vec{e}$. This chapter completes the algorithm's description by presenting the heuristic function, F(), that is used to navigate the rule space. If function F() determines that rule r'_i is a better candidate of specialization than rule r'_j, then it will return numerical values, such that $\mathtt{F}(r'_i, r) > \mathtt{F}(r'_j, r)$, where r is the parent rule to both r'_i and r'_j.

Three enhancements to the heuristic function are presented in this chapter:

- The parent-child approach, implicitly proposed by LazyDT, is explicitly investigated so that its utility may be empirically evaluated in Chapter 7.
- A simple pruning mechanism is integrated to mitigate against overfitting, and therefore improve accuracy.
- Three different base measures are demonstrated to help locate an accurate version of the function.

6.1 Information Available to F()

Before proceeding with a detailed description of the several versions of F() that will be proposed in this chapter, the information that is available for a measure to make a heuristic selection is presented. Recall that two parameters are passed to $\mathtt{F}(r', r)$: the current rule r and its proposed specialization r'. From these two parameters, the proposed heuristic measures may make use of five derived pieces of information:

1. $\vec{r'}$: cpd.[1] vector for the consequent of rule r'
2. $\vec{r}$: cpd. vector for the consequent of rule r
3. $\vec{r'}_c$: cpd. vector for the records covered by r but not covered by r'.
4. p': proportion of the records covered by r that are also covered by r'.
5. p'_c: proportion of the records covered by r that are not covered by r'.

The following example presents the derivation of each of the five pieces of information for a specific call to F(). This example will also be used in the detailed descriptions of the different versions of F().

Example 6.1. This example is based on the call to F(r_2,r_0) in Example 4.1. Recall the contents of the two rules:

$$r_0: \quad (A_1 = v_{any}) \wedge (A_2 = v_{any}) \wedge (A_3 = v_{any}) \quad \rightarrow A_4 \in [c_1 = 30, c_2 = 70]$$
$$r_2: \quad (A_1 = v_{any}) \wedge (A_2 = v_2) \wedge (A_3 = v_{any}) \quad \rightarrow A_4 \in [c_1 = 14, c_2 = 6]$$

From these two rules the *class probability distribution vectors* for each rule consequent may be derived by dividing each class summary by the cover of each rule.

$$\vec{r_0}: \quad [30/(30+70), 70/(30+70)] \quad = [0.3, 0.7]$$
$$\vec{r_2}: \quad [14/(14+6), 6/(14+6)] \quad = [0.7, 0.3]$$

A class probability distribution vector $\vec{\alpha}$ is constrained by

$$1) \quad 0 \leq \alpha_i \leq 1 \tag{6.1}$$
$$2) \quad \sum_{i=0}^{c} \alpha_i = 1 \tag{6.2}$$

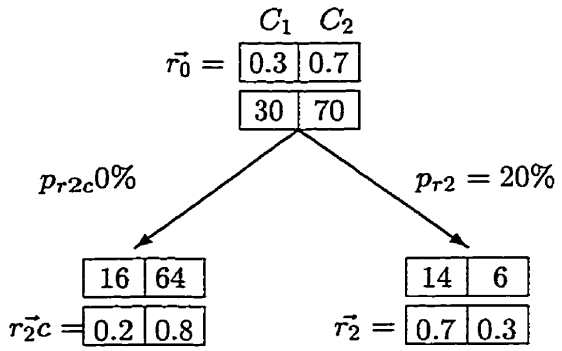
where α_i is the value of the ith element and c is the number of distinct classes in A_c.

The third piece of information to be derived, is the class distribution of the records covered by parent rule r but which are not covered by rule r'. These records represent the records covered by the complement rule r'_c which negates the proposition which r' specialized on. The complement rule for our example above is:

$$r_{2c}: \quad (A_1 = v_{any}) \wedge (A_2 \neq v_2) \wedge (A_3 = v_{any}) \quad \rightarrow A_4 \in [c_1 = 30 - 14, c_2 = 70 - 6]$$
$$\rightarrow A_4 \in [c_1 = 16, c_2 = 64]$$

[1] class probability distribution

Figure 6.1: Graphical presentation of the information available to the different versions of F(), to evaluate the predictive value of specializing from rule r_0 to rule r_2, in Example 4.1.



The final items of interest are the proportions of records covered by the parent rule that are covered by both sibling rules: r', r'_c. These proportions will be referred to as p'_r and p'_{rc}. For our example the following values would apply:

$$p_{r2} \Leftarrow \frac{14+6}{30+70} = 20/100 = 0.20$$
$$p_{r2c} \Leftarrow \frac{16+64}{30+70} = 80/100 = 0.80$$

Figure 6.1 summarizes the information available for $\mathtt{F}(r_2, r_0)$.

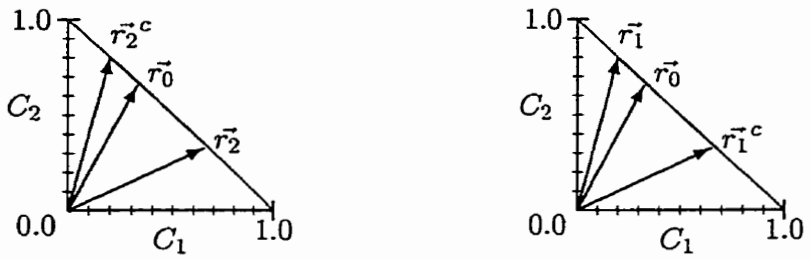
□

6.2 Sibling-Sibling versus Parent-Child

The standard approach to heuristic function calculation in top-down induction is to determine the difference between the class probability distribution vector of the two sibling rules: r', r'_c [52, 57]. In our example above this is the difference between $\vec{r_2}$ and its complement vector $\vec{r}_{2c}$. This approach will be referred to as *sibling-sibling* functions. A variation to this approach, informally presented for LazyDT, is to evaluate the difference between the parent rule r and the sibling rule that applies to the event vector, r'. In the case of our example, this is represented by the difference between $\vec{r_0}$ and $\vec{r_2}$. This approach will be referred to as *parent-child* functions.

One possible intuition for the use of the parent-child variation, is that a large difference

Figure 6.2: Graphical presentation of the differences between class probability distribution vectors. The figure on the left represents the information for $\mathtt{F}(r_2, r_0)$ already presented in Figure 6.1. The figure to the right represents the information that would be generated for $\mathtt{F}(r_1, r_0)$. Each dimension represents the proportion among the two class values. The line drawn diagonally between $(1, 0)$ and $(0, 1)$ is a reminder that the components of each vector add up to 1 (100%).



between siblings may still occur even though the parent and child vectors may be rather similar. Figure 6.2 graphically presents this situation by comparing the calculation of $\mathtt{F}(r_2, r_0)$ and $\mathtt{F}(r_1, r_0)$. Because the sibling-sibling differences for these two distributions are identical, the call to $\mathtt{F}(r_2, r_0)$ will return a value equal to $\mathtt{F}(r_1, r_0)$, if F() measures sibling-sibling differences. However, if F() measures parent-child differences, $\mathtt{F}(r_2, r_0)$ will be greater than $\mathtt{F}(r_1, r_0)$.

6.3 Sibling-Sibling F()

The approach used by eager top-down induction algorithms to calculate F(), is to measure the difference between siblings. This approach has been extensively researched for top-down induction of decision trees [57, 9]. The *sibling-sibling* versions of F(), based on the *entropy*(), $ORT()$ and $DI()$ measures, are reviewed below along with sample results.

6.3.1 Average Impurity *entropy()*

The entropy function $i()$, which is based on information theory, is commonly used to base decisions on which path to follow [57, 65].

$$i(\vec{\alpha}) = -\sum_{j=1}^{c} \vec{\alpha}_j \log_2 \vec{\alpha}_j \tag{6.3}$$

This function measures the impurity of a class probability distribution vector. If the class probabilities are evenly distributed among the c class values, the vector is referred to as impure. If the distribution is skewed to a single class, and therefore makes a very clear prediction, the vector is referred to as pure. The entropy for the three class probability vectors in Example 6.1 are:

$$\begin{aligned} i(\vec{r_0}) &= 0.88 bits \\ i(\vec{r_2}) &= 0.88 bits \\ i(\vec{r_{2c}}) &= 0.72 bits \end{aligned}$$

$i(\vec{r_0}) = i(\vec{r_2})$ because their class probability distributions have the same proportions, and the order does not impact the measure.

The sibling-sibling heuristic based on this measure, evaluates the average purity (information) gain for the sibling vectors, $\Delta i()$. The more the sibling vectors make clearer predictions than the parent rule, the higher the value it gives to a specialization step. This heuristic will be informally referred to as *entropy*(). For our example, the information gain is determined by subtracting the average entropy of the sibling rules from the entropy of the parent rule:

$$\begin{aligned} \Delta i(\vec{r_0}, \vec{r_2}) &= i(\vec{r_0}) - p_{r2} i(\vec{r_2}) - p_{r2c} i(\vec{r_{2c}}) \\ &\approx 0.88 - 0.2(0.88) - 0.8(0.72) \\ &\approx 13\text{bits} \end{aligned}$$

If no other tests achieves an information gain that is greater than or equal to this value of 0.13 bits then this test will be selected.

6.3.2 Angle-based Measure $ORT()$

Another measure previously used as a heuristic to guide hypothesis construction is the angle between class probability distribution vectors [26]. Rather than determine the angle however, the $ORT()$ Function 6.4, subtracts 1.0 from the cosine of the angle θ between the two vectors.

$$ORT(\vec{\alpha}, \vec{\beta}) = 1 - \cos\theta(\vec{\alpha}, \vec{\beta}) \tag{6.4}$$

The reason that the *cosine* function is selected over the angle θ between two vectors is likely because of the simple computation of this function based on the inner (dot) product[2] $\vec{\alpha} \circ \vec{\beta}$ and each vector's Euclidean distance[3] $\|\vec{\alpha}\|$.

$$\cos\theta(\vec{\alpha}, \vec{\beta}) = \frac{\vec{\alpha} \circ \vec{\beta}}{\|\vec{\alpha}\| \cdot \|\vec{\beta}\|} \tag{6.5}$$

Also, by subtracting 1, the bounds of this formula are [0,1], where 1 represents the greatest separation (orthogonality) between two class distribution vectors and 0 the least.

For Example 6.1, $ORT(\vec{r}_2, \vec{r}_{2c})$, results in:

$$\begin{aligned} ORT(\vec{r}_2, \vec{r}_{2c}) &= 1 - \cos\theta\Big([.7,.3],[.2,.8]\Big) \\ &= 1 - \frac{(.8)(.3)+(.2)(.7)}{(\sqrt{.7^2+.3^2})(\sqrt{.2^2+.8^2})} \\ &\approx 1 - \frac{0.38}{0.63} \\ &\approx 1 - 0.61 = 0.39 \end{aligned}$$

6.3.3 Normalized Geometric Distance $DI_n()$

An alternative to measuring the angle between two class probability distribution vectors, is to measure the geometric distance between them, $DI(\vec{\alpha}, \vec{\beta})$. This measure is a component of the measure used by the InferRule decision tree algorithm [67]. The range for this function, however is demonstrated to be $[0, \sqrt{2}]$. For DBPredictor, the $DI_n()$ function is proposed which normalizes the range to $[0, 1]$, to behave similarly to the $ORT()$ class separation measure.

The Raw Distance Measure: $DI()$

The Euclidean distance of two class probability distribution vectors $\vec{\alpha}$, $\vec{\beta}$ is:

$$DI(\vec{\alpha}, \vec{\beta}) = \sqrt{\sum_{i=1}^{c} (\alpha_i - \beta_i)^2} \tag{6.6}$$

[2]Where $\vec{\alpha} \circ \vec{\beta} = \sum_{i=0}^{c} \alpha_i \beta_i$

[3]Where $\|\vec{\alpha}\| = \sqrt{\sum_{i=0}^{c} \alpha_i^2}$

For Example 6.1, a call to $DI(\vec{r}_2, \vec{r}_{2c})$, results in:

$$\begin{aligned} DI(\vec{r}_2, \vec{r}_{2c}) &= \sqrt{\sum_{i=1}^{c} (r\vec{2},i - r\vec{'}_{2,i})^2} \\ &= \sqrt{(.2-.7)^2 + (.8-.3)^2} \\ &= \sqrt{0.5} \approx 0.71 \end{aligned}$$

This distance can be visually validated against Figure 6.2.

The Range of $DI()$

To compose the $DIn()$ function, the range for $DI()$ is now investigated.

Proposition 6.1. Given two class probability distribution vectors $(\vec{\alpha}, \vec{\beta})$ the minimum value of $DI(\vec{\alpha}, \vec{\beta})$ is 0

Proof. The minimum distance occurs when the two vectors are identical. When the two vectors are identical their distance is 0, as desired. □

Proposition 6.2. Given two class probability distribution vectors $\vec{\alpha}$, $\vec{\beta}$, the $max(DI(\vec{\alpha}, \vec{\beta})) = \sqrt{2}$

Proof. First we show that for any c there is always be $\vec{\alpha}, \vec{\beta}$ such that $DI(\vec{\alpha}, \vec{\beta}) = \sqrt{2}$. Then we show that that the measure can be no greater than $\sqrt{2}$.

1. In a two (c=2) dimensional problem, a distance of $\sqrt{2}$ is achieved with $\vec{\alpha} = [1, 0]$ and $\vec{\beta} = [0, 1]$. If we extend this example into n dimensions so that each new dimension contains the value 0 the function will continue to be equal to $\sqrt{2}$. Thus there is always an instance in which $DI(\vec{\alpha}, \vec{\beta}) = \sqrt{2}$.

2. The definition for $DI()$ (6.6) can be expanded to 6.7. We will show that that the two positive terms in the expansion ($\sum_{i=1}^{c} \alpha_i^2$ and $\sum_{i=1}^{c} \beta_i^2$) must each be ≤ 1. Recall that by the constraints on class probability distribution vectors, $\alpha_i \leq 1$ and thus $\alpha_i^2 \leq \alpha_i$. Further, $\sum_{i=1}^{c} \alpha_i = 1$ and thus $\sum_{i=1}^{c} {\alpha_i}^2 \leq 1$.

$$\sqrt{\sum_{i=1}^{c} \alpha_i^2 + \sum_{i=1}^{c} \beta_i^2 - \sum_{i=1}^{c} 2\alpha_i\beta_i} \tag{6.7}$$

Because the third term is the expansion can only subtract from the total, we assume that it is minimized to 0. Thus the expansion must be $\leq \sqrt{1+1-0} \leq \sqrt{2}$

To summarize, $DI(\vec{\alpha}, \vec{\beta}) \leq \sqrt{2}$ and regardless of the number of dimensions (c) there is always a combination in which $DI(\vec{\alpha}, \vec{\beta}) = \sqrt{2}$. Thus $max(DI(\vec{\alpha}, \vec{\beta})) = \sqrt{2}$, as desired. □

The Normalized Distance Measure: $DI_n()$

Given the range of $[0, \sqrt{2}]$ for $DI()$, the $DI_n()$ distance function is composed by a simple transformation.

$$DI_n(\vec{\alpha}, \vec{\beta}) = \frac{DI(\vec{\alpha}, \vec{\beta})}{\sqrt{2}} \tag{6.8}$$

This transformation will facilitate the interchange of $DI_n()$ within an algorithm that already makes use of the $ORT()$ measure. For Example 6.1, a call to $DI_n(\vec{r}_2, \vec{r}_{2c})$, results in:

$$\begin{aligned} DI_n(\vec{r}_2, \vec{r}_{2c}) &= \frac{DI(\vec{r}_2, \vec{r}_{2c})}{\sqrt{2}} \\ &= \frac{\sqrt{1/2}}{\sqrt{2}} = 1/2 \end{aligned}$$

6.4 Parent-Child F()

Eager top-down classifiers must test for *sibling-sibling* differences because they do not know which path a particular event will take. In lazy model-based induction, however, the use of the complement rule r'_c to estimate a specialization step's predictive value appears to be artificial. It may instead be more valid to focus attention on the different between the consequents of the parent rule and the sibling rule that applies to $\vec{e}$. In Example 6.1, this is the difference between $\vec{r}_0$ and $\vec{r}_2$. The use of this variation is implicitly proposed for the LazyDT classifier [29] with the use of the *entropy()* measure (see Section 6.4.1 below). To test the validity of this approach, this section explicitly presents the parent-child variation

for the three base measures already presented for the sibling-sibling definitions of F(). These updated functions will be represented with a subscripted + symbol. In the next section the $entropy_+()$, $ORT_+()$ and $DI_+()$ are described. These two final variations have not been presented before but they are simply based respectively on the angle and distance between the class probability distributions of r and r'.

6.4.1 entropy$_+$() Variation

As proposed for LazyDT [29], the *parent-child* variation of the entropy based function $\Delta i_+()$, subtracts the entropy $i()$ of the parent rule from the entropy of the child rule. Informally, this function will be referred to as $entropy_+()$.

$$\Delta i_+(\vec{\alpha}, \vec{\beta}) = i(\vec{\alpha}) - i(\vec{\beta}) \tag{6.9}$$

Unfortunately, as documented in [29], this simplistic approach leads to problems. Our example highlights this problem because the entropy measures of both parent and child distributions are equal and cancel themselves out.

$$\begin{aligned} \Delta i_+(\vec{r_0}, \vec{r_2}) &= i(\vec{r_0}) - i(\vec{r_2}) \\ &= 0.88 - 0.88 = 0 \end{aligned}$$

To overcome this problem the distribution of r_0 is flattened (made impure) and the distribution of r_2 is updated accordingly. This normalization process is not described in detail in [29]. We assume the following process for parent rule α and child rule β:

$$\begin{aligned} \vec{\alpha}_+ \Leftarrow \quad \forall i[\vec{\alpha}_{i+}] &= 1/c \\ \vec{\beta}_+ \Leftarrow \quad \forall i[\vec{\beta}_{i+}] &= \frac{\vec{\beta}_i/\vec{\alpha}_i}{\sum_{j=1}^{c} \vec{\beta}_j/\vec{\alpha}_j} \end{aligned}$$

When applied to Example 6.1, r_0 and r_2 are updated to:

$$\begin{aligned} \vec{r_{0+}} &= [1/2, 1/2] = [0.5, 0.5] \\ \vec{r_{2+}} &= \left[\frac{.7/.3}{.7/.3 + .3/.7}, \frac{.3/.7}{.7/.3 + .3/.7}\right] \\ &\approx [0.84, 0.16] \end{aligned}$$

Now rather than canceling itself out, $entropy_+()$ achieves a positive value

$$\begin{aligned} \Delta i_+(\vec{r_{0+}}, \vec{r_{2+}}) &\approx i([0.5, 0.5]) - i([0.84, 0.16]) \\ &\approx 1 - 0.63 = 0.37 \end{aligned}$$

6.4.2 $\mathbf{ORT_+()}$ Variation

The parent-child variation of the $ORT()$ based function, simply measures the angle between the class probability distribution of the parent rule and the child rule. For parent and child class distributions $\vec{\alpha}$, $\vec{\beta}$

$$ORT_+(\vec{\alpha}, \vec{\beta}) = 1 - \cos\theta(\vec{\alpha}, \vec{\beta}) \tag{6.10}$$

For Example 6.1, a call to $ORT_+(r_0, r_2)$, results in:

$$\begin{aligned} ORT(\vec{r_0}, \vec{r_2}) &= 1 - \cos\theta\Big([.3, .7], [.7, .3]\Big) \\ &= 1 - \frac{(.7)(.3) + (.3)(.7)}{(\sqrt{.7^2 + .3^2})(\sqrt{.3^2 + .7^2})} \\ &\approx 1 - \frac{0.42}{0.58} \\ &\approx 1 - 0.72 = 0.28 \end{aligned}$$

6.4.3 $\mathbf{DI_+()}$ Variation

The parent-child variation of the $DI()$ based heuristic function, simply measures the geometric distance between the class probability distribution of the parent rule and the child rule. For parent and child class distributions $\vec{\alpha}$, $\vec{\beta}$

$$DI_+(\vec{\alpha}, \vec{\beta}) = \sqrt{\sum_{i=1}^{c} (\vec{\alpha_i} - \vec{\beta_i})^2} \tag{6.11}$$

For Example 6.1, the Euclidean distance between r_0 and r_2 is

$$\begin{aligned} DI_+(\vec{r_0}, \vec{r_2}) &= \sqrt{\sum_{i=1}^{c} (\vec{r_{0i}} - \vec{r_{2i}})^2} \\ &= \sqrt{(.3 - .7)^2 + (.7 - .3)^2} \\ &= \sqrt{0.32} \approx 0.57 \end{aligned}$$

To normalize the $DI_{+}()$ function, the $DI_{n+}()$ divides the result by $\sqrt{2}$.

$$DI_{n+}(\vec{\alpha}, \vec{\beta}) = \frac{DI_{+}(\vec{\alpha}, \vec{\beta})}{\sqrt{2}} \tag{6.12}$$

6.5 Pruning

A general characteristic to induction algorithms in real-world domains, is their attempt to fit the noise in the dataset into their model rather than finding the true model. This phenomenon is often referred to as "overfitting" (or overspecialization) [18, 68]. A common method to counteract overfitting, is the addition of various penalty terms to the heuristic function, sometimes referred to as "pruning" criteria. Because DBPredictor is found to be vulnerable to overfitting in Chapter 7, a simple pruning mechanism is proposed to initiate the investigation in this area.

This section proposes two additional criteria for the heuristic function F() that may help to avoid basing prediction on unreliable rules and that will avoid searching uninteresting terrain. This approach is similar to the use of pre-pruning (also known as stop-splitting rules) used in top-down induction of decision trees [52, 58, 67]. Specifically, further specialization will be restricted to rules that match a minimum number of instances in the dataset, and to rules that achieve a minimum heuristic function F() value. To achieve this, two internal parameters are integrated into F(): `min_cover` and `min_value`. The addition of these two tests will increase the running time complexity by a constant. However, because these additions stop specialization from proceeding, some effort is saved on average. Informal investigation has shown that the increased running time due to the updates is generally negated by the shortening of the search space. Values for these two internal parameters will be investigated in the empirical studies of Chapter 7.

6.5.1 `min_cover` threshold

The minimum rule cover threshold parameter (`min_cover`) describes the smallest number of records that a rule must cover. The intuition behind this update is to reject rules based on too few instances as sensitive to noise and therefore unreliable. This parameter must have

a value within the range of $[1, m]$. At a setting of 5, for example, each specialization step will determine whether the cover of the proposed rule r' is ≥ 5. If rule r' does not achieve this threshold its evaluation will be set to zero by F().

$$|r'_{cover}| < \texttt{min\_cov} \rightarrow \texttt{F}(r', r) = 0$$

6.5.2 min_value threshold

The minimum heuristic function value threshold parameter (min_value) describes the smallest heuristic function value that a rule must achieve. The intuition behind this update is to terminate a specialization path if it does not make significant progress. This parameter must have a value that is greater than or equal to 0 (no pruning) and less than or equal to 1 (complete pruning). At a setting of 0.10, for example, each specialization step must attain a heuristic value ≥ 0.10. If this is not the case, then the value assigned to the specialization is set 0.

$$\texttt{F}(r', r) < \texttt{min\_meas} \rightarrow \texttt{F}(r', r) = 0$$

6.6 Chapter Summary

This chapter presents the definition of the heuristic function F() that is used by DBPredictor to navigate the rule space. Three enhancements are proposed to the function. First, the parent-child calculation method, implicitly used by LazyDT, is explicitly described. The focus on the difference between parent and child class distribution vectors may prove to be more accurate. Second, a simple pruning mechanism is integrated into F(). In other top-down induction algorithms, the use of pruning has helped to mitigate against overfitting. Finally, three base measures are implemented for the function. With this level of option and variety, a more accurate measure may appear. Part of the scope in the next chapter, is an empirical assessment of which version of F() achieves good accuracy. Subgoals to this assessment are to determine whether the *parent-child* variation results in superior accuracy and whether simple pruning mitigates against overfitting.

Chapter 7

Empirical Study of Accuracy

This chapter presents the results of an empirical study into DBPredictor's accuracy. The main question addressed within the study was DBPredictor's suitability for on-line classification tasks, with respect to accuracy. Because these tasks have a prevalence of irrelevant attributes and underspecified events, DBPredictor's relative performance within these types of domains was of key importance. In summary, these tests show that while IB1's accuracy degraded in the presence of irrelevant attributes and C4.5's accuracy degraded with underspecified events, DBPredictor tied for the top position under both tests. This result presents evidence that the algorithm is particularly suited to on-line classification tasks.

A few other secondary questions are addressed within this study to better understand the response of DBPredictor's accuracy with respect to

- pruning
- overspecialization
- parent-child versus sibling-sibling calculation
- different base measures: $entropy()$, $ORT()$, $DI_n()$
- numerical attribute handling

The chapter is organized into three sections. Section 7.1 presents the methodology used within the study to validate statements of accuracy. Section 7.2 reports the studies used to locate an accurate version of DBPredictor. Within this section the role pruning, parent-child and different base measures in F() are investigated. Once a relatively accurate version of

DBPredictor was identified, Section 7.3 reports DBPredictor's relative performance, when compared to the IB1 and C4.5 algorithms. Within this section the role of overspecialization and numerical attribute handling are also investigated.

7.1 Methodology

This section describes the methodology that was used to support the hypothesis that a particular classification algorithm was more accurate than another. While there is no standard empirical method to accomplish this, some general guidelines are available. First a group of representative datasets are gathered, next an error rate estimation method is selected and finally a set of criteria are defined that must be passed before the hypothesis is confirmed or rejected [21, 41, 68]. To follow these guidelines, Section 7.1.1 reports the group of representative datasets. Section 7.1.2, report the error rate estimation method used. And, Section 7.1.3 presents the criteria selected to test a hypothesis on accuracy.

7.1.1 Datasets

Twenty three real-world data sets were selected to perform this study's investigation into DBPredictor's accuracy. An attempt was made to include datasets that have been widely used in other studies of misclassification rates [21, 29, 60]. Furthermore, the datasets were required to contain a variety of sizes, data types, and application areas. Table 7.1 lists and summarizes the characteristics of these datasets. The *anneal* dataset (AN), for example, is shown to contain $n = 898$ records and $m = 39$ attributes (9 of which are numerical). The class attribute for this dataset contains six unique classes and the most common class appears 76.2% of the time. Finally 64.8% of the dataset's values are missing. All datasets were retrieved from the UCI repository [51] at:

`ftp://ics.uci.edu/pub/machine-learning-databases`.

7.1.2 Error Rate Estimation

A classification algorithm's accuracy is commonly represented by the percent of incorrectly predicted event vectors (*error rate*). To ensure that good *estimates* of the true error rate are

Table 7.1: Characteristics of the datasets used in the empirical study of DBPredictor's accuracy. The columns are: row number; common name; number of instances (n); number of predicting attributes ($m-1$) and numerical attributes (num); number of classes and the percentage of the most common class; and percentage of missing values. Datasets with an * beside their common name were used to fine tune DBPredictor's and IB1's threshold parameters.

#	Dataset	abbrv.	n	$m-1/num$	Clss./Def%	Miss.%
1	anneal	AN	898	38/9	6/76.2	64.9
2	audiology	AD	226	69/0	24/25.3	2.1
3	breast-w	BW	699	10/9	2/65.5	0
4	chess	CH	3196	36/0	2/52.2	0
5	credit-a	CA	690	14/6	2/55.5	0
6	credit-g	CG	1000	20/7	2/70.0	0
7	diabetes	DI	768	8/8	2/65.1	0
8	echocardiogram*	EC	132	7/6	3/81.8	4.6
9	glass	GL	214	10/9	7/35.5	0
10	hayes-roth*	HR	160	4/0	3/40.6	3
11	heart*	HT	270	13/7	2/55.6	0
12	heart-c	HC	303	13/7	5/54.1	0
13	heart-h	HH	94	13/7	2/63.9	0
14	hepatitis	HE	155	19/6	2/79.4	5.7
15	horse-colic	HO	368	22/10	2/63.0	0
16	iris*	IR	150	4/4	3/33.3	0
17	letter	LT	20000	16/16	26/4.1	0
18	liver-disease	LD	345	6/6	2/58.0	0
19	mushroom	MU	8124	22/0	2/51.8	0
20	segment	SE	2310	19/19	7/14.3	0
21	soybean-small	SO	47	35/0	4/36.2	0
22	tic-tac-toe*	TO	958	9/0	2/65.3	0
23	vote	VO	493	16/0	2/61.4	0

reported, all error rate results in this study are based on the average of at least five runs of the stratified 10-fold cross-validation (SCV-10) resampling technique. SCV-10 has been used extensively in past empirical studies [9, 58]. The averaging of several cross-validation runs has been recently proposed to overcome the large variance of resampling techniques [43, 68]. This approach to estimation tends to achieve a low and conservative bias and variance [43].

SCV-10 requires that the examples be randomly allocated to 10 mutually exclusive partitions of approximately equal size, while maintaining approximately the same class distribution as in the original data set. With SCV-10 all the dataset records contribute to the estimate and almost all the cases (90%) are used to base each prediction. For example, a domain with 101 cases and a 60%/40% distribution of its binary (0,1) class label would result in nine partitions with 10 cases and one partition with 11 cases. Each partition would have approximately six cases with class label 0 and four cases with class label 1. Once the partitioning has completed, ten tests are performed. Each partition will at one point be labeled the *test set* while the remaining nine partitions are grouped together into the *training set*. Within each of the ten tests, every record from the test set is transformed into an event vector $\vec{e}$ and a prediction is made based on the examples stored in the *training set*. The ratio of misclassified cases is recorded for each of the ten tests. These ten ratios are averaged to give the cross-validated error rate. For this study at least five SCV-10 were performed on each dataset to achieve an average error rate estimate along with its standard deviation σ.

7.1.3 Hypothesis Testing Criteria

Given a method to estimate an algorithm's average error rate, what is required to claim that one algorithm is more accurate than another? Just as there is no standard list of datasets that must be tested there is currently no single comparative measure of accuracy. Commonly however, several individual measures are used in combination. Only when an algorithm succeeds on all the tests can this algorithm be declared more accurate [21].

For this study a classification algorithm A_1 will be referred to as more accurate than classifier A_2, if it meets the following criteria:

- **lower on**: A_1 achieves a lower error rate on more datasets than A_2.
- **avg**: A_1 achieves a lower average error rate than A_2.

Table 7.2: Example of one algorithm (A_1) being more accurate than another (A_2).

F()	lower on	avg	clear wins
A_1	3	15.20%	2
A_2	2	16.54%	0
Subtract	1	-1.34%	2

- **clear wins:** A_1 achieves a lower error rate, with a 99.5% confidence level, on more datasets than A_2 (based on a one-tailed t-test [34]).

Table 7.2 reports a sample result were algorithm A_1 is more accurate than A_2. Based on this report algorithm A_1 can be said to be more accurate then algorithm A_2. First, A_1 achieved a lower error rate on one more dataset than A_2. Second, the average error rate over all datasets is lower for A_1 (by 1.34%). Finally, based on a one-tailed *t-test*[1], algorithm A_1 was more accurate over A_2 with a 99.5% confidence on two more datasets. If all three criteria had not been met, then no claim of accuracy superiority could be made.

7.2 Variations on F()

The focus of the first set of empirical investigations into DBPredictor's accuracy, was to discover an accurate combination of heuristic function and threshold parameter settings. This involved testing a large number of versions of F(). The results from this investigation presented an opportunity to also investigate the following questions:

1. What is the impact of the *parent-child* calculation approach on the algorithm's accuracy?

2. What is the impact of pruning on the algorithm's accuracy?

3. Which heuristic function results in the highest accuracy?

[1] $T = (\overline{X} - \overline{Y}) / \sqrt{(\frac{1}{n_1} + \frac{1}{n_2}) \frac{(n_1-1)S_X^2 + (n_2-1)S_Y^2}{n_1+n_2-2}}$ [34]

The version of DBPredictor which achieved the highest accuracy is then used by the next phase of the study which tests the algorithm's relative performance.

7.2.1 Datasets

A small group of datasets were selected from Table 7.1 to discover an accurate version of DBPredictor. The use of smaller group of datasets allowed for the testing of a large number of heuristic functions and threshold parameter combinations. More importantly, this approach also left the optimization of DBPredictor blind to the majority of datasets.

The five selected datasets were: *echocardiogram*, *hayes-roth*, *heart*, *horse-colic*, and *iris* datasets. These datasets (marked in Table 7.1 with a * symbol beside their name) contain a sampling of attribute types and domains. For this initial study however the datasets needed to be small enough (≈ 400) to facilitate an intensive study of the different heuristic measures and threshold parameter settings. Approximately seven million predictions were performed to analyze the algorithm's response! Any increase to the number of datasets or their average size would have increased not only the number of predictions, but also the time required by each prediction. It is difficult to determine whether the five selected datasets are unbiased and representative. However, the significant differences of optimal parameters settings encountered within the study for each of the datasets, indicate that the set may have been appropriate.

7.2.2 Threshold Setting Refinement

DBPredictor contains three threshold parameters. In Chapter 4 the numerical partitioning ratio (`num_part`) was defined to handle numerical attributes. In Chapter 6, the minimum rule cover (`min_cover`), and minimum evaluation function value (`min_meas`) parameters were defined to implement pruning support. For these three numeric parameters, a range of values were selected based on preliminary experiments. The final criteria for a valid parameter settings combination was that none of its settings could could be a terminal value. For a setting of 1.50 to be recommended for `num_part`, for example, values greater than 1.50 (e.g. 2.00) and less than 1.50 (e.g. 1.25) must have also been investigated. The following list describes the range of settings selected for the algorithm's three internal parameters:

- The numerical partitioning ratio parameter (`num_part`) determines how aggressively to divide the region around a numerical value on each specialization step. This parameter

must be set with a value that is greater than 1.0. At a setting of 2.00, for example, each specialization step will divide the matching range around a value by two. After some preliminary experimentation, the settings selected for the num_part parameter were: **1.25, 1.50, 2.00, 4.00** and **6.00**.

- The minimum rule cover threshold parameter (min_cover) is a pruning based stopping criteria used to mitigate from overspecialization. A setting for this parameter describes the smallest number of records that may be covered by a rule. This parameter must have a value within the range of $[0, n]$. At a setting of 5, for example, each specialization step will determine whether the cover of the proposed rule is ≥ 5. After some preliminary experimentation, the settings selected for the min_cover parameter were: **1, 2, 5,** and **10**.

- The minimum heuristic function value threshold parameter (min_meas), is another pruning stopping criterion used to mitigate against overspecialization. A setting for this parameter describes the smallest heuristic function value allowed between specialization steps. This parameter must have a value that is greater than or equal to 0 (no pruning) and less than or equal to 1 (complete pruning). At a setting of 0.10, for example, each specialization step will determine whether the class separation measure for the proposed new rule is ≥ 0.10. If this is not the case then this proposed specialization is rejected. After some preliminary experimentation, the settings selected for the min_meas parameter were: **0.00, 0.01, 0.05, 0.10, 0.33,** and **0.50**.

To determine which parameter settings achieved near optimal accuracy for each of the six heuristic functions, all parameter combinations were tested against each of the five datasets. The different settings proposed above meant that 120 parameter combinations were possible ($5 \times 4 \times 6$). Table 7.3 shows the results of DBPredictor's accuracy when the $DI_n()$ version of the algorithm was tested against the *iris* dataset. To attain the lowest error rate against this dataset, with this algorithm, the following parameter settings are required: num_part=6.00, min_cover=1 and min_meas=0.33. After five 10-SCV studies, this parameter combination resulted in a 3.53% average error rate.

Each of the five datasets required a different parameter setting combination to minimize the error rate achieved by each heuristic function. Table 7.4 shows the parameter combinations that achieved the lowest error rate for the $DI_n()$ based algorithm. The optimal

Table 7.3: Average error rate on the *iris* dataset for DBPredictor based on the $DI_n()$ heuristic function. For conciseness only the top five and the bottom ranked combinations are presented. The fifth combination happens to perform no pruning (min_cover=1 and min_meas=0.00.

#	min_cover	min_meas	num_part	err. rate↓
1	1	0.33	6.00	3.53%
2	10	0.20	6.00	3.65%
3	5	0.20	6.00	4.00%
4	10	0.10	4.00	4.00%
5	1	0.00	1.50	4.10%
...	...	...	...	...
120	5	0.50	1.25	10.68%

Table 7.4: Internal parameter settings for the $DI_n()$ based DBPredictor that resulted in the lowest error rate for each of the five datasets. The bottom row reports the average error rate over the five datasets.

#	Dataset	min_cover	min_meas	num_part	err. rate↓
1	iris	1	0.33	6.00	3.53%
2	hayes-roth	1	0.20	2.00	13.13%
3	horse-colic	10	0.50	6.00	15.80%
4	heart	10	0.00	2.00	17.60%
5	echocardiogram	10	0.33	1.50	18.00%
Average					13.61%

combination for the *heart* dataset, for example, is significantly different than the optimal combination already encountered for the *iris* dataset. In the case of the $DI_n()$ heuristic function the average error rate with the optimal settings on each dataset was 13.61%. Similar differences in optimal parameter setting combinations occurred with the other five heuristic functions.

Because of the large variety of optimal parameter combinations, a compromise was required to select a single combination that minimizes this error rate. One way to determine which combination to chose, is to select the combination that was more accurate than all other combinations. However, because the parameter settings were set intentionally close, no single combination was able to meet the required criteria. Between the top ranked

Table 7.5: Average error rate over the five selected datasets achieved by the $DI_n()$ function at particular combinations. For conciseness only the top five, the top unpruned and the least accurate combination are presented.

#	min_cover	min_meas	num_part	err. rate ↓
1	5	0.10	1.50	15.13%
2	5	0.05	1.50	15.30%
3	5	0.10	2.00	15.33%
4	5	0.10	1.25	15.51%
5	5	0.05	1.25	15.52%
...	...	...	...	...
33	1	0.00	1.50	18.23%
...	...	...	...	...
120	2	0.50	1.25	27.40%

Internal parameter combinations that achieve the lowest average error rate for each of the six heuristic functions over the five datasets.

F()	min_cover	min_meas	num_part	err. rate ↓
$DI_n()$	5	0.10	1.50	15.13%
$entropy()$	5	0.01	1.50	15.18%
$ORT()$	5	0.00	1.50	15.30%
$DI_{n+}()$	5	0.05	2.00	15.33%
$entropy_+()$	5	0.01	2.00	16.54%
$ORT_+()$	5	0.00	2.00	16.72%

combinations, for example, none achieved a lower average error rate over the five datasets with 99.5% confidence.

The selection of an accurate combination was instead based on the combination that strictly minimized the average error rate over these five data sets. Table 7.5 shows that the values [min_cov=5, min_meas=0.10, num_part=1.50] achieve this for the $DI_n()$ heuristic function. With this parameter setting combination and this heuristic function, the average error rate of 15.13% was smaller than any other tested combination.

After a similar exercise was performed on the remaining versions of the algorithm, the optimal parameter setting combinations were located for each of the six heuristic functions. Table 7.2.2 presents these results. While more optimal setting combinations likely exist, the general region of these settings has likely been located.

Observations

Some initial observations can be made from the optimization results reported in Table 7.2.2 with respect to the three questions relevant to this section. The most striking feature about the table is that all algorithms selected a value greater than 1 for a minimum cover. This points to the possibility that pruning does indeed improve accuracy. The next section, Section 7.2.3, will address this question further. A second observation is that the algorithms which used the *parent-child* heuristic function variation (marked with a $_+$) achieved higher error rate than all *sibling-sibling* versions. This observation will further analyzed in Section 7.2.4. Finally the $DI_n()$ heuristic function achieves the lowest error rate (15.13%). Section 7.2.5 will discuss whether this heuristic function does indeed generate the most accurate version of DBPredictor based on the five datasets.

7.2.3 Pruning's Impact on Accuracy

Because all six heuristic functions made use of pruning to achieve their lowest error rate, it appears promising that pruning does indeed generate a more accurate version of DBPredictor. To test this hypothesis, the algorithms with the optimal parameter settings were compared to the versions of the algorithm that did not make use of pruning. To turn off pruning the `min_cover` parameter was set to 1 and the `min_meas` parameter was set to 0.00. With these two parameters fixed, the only setting left to optimize was for the `num_part` parameter. Table 7.6 presents the result of this investigation.

Table 7.6: *num_part* settings that achieve the lowest average error rate for each of the six heuristic functions when pruning is turned off ($min\_cover = 1$, $min\_meas = 0.00$).

F()	min_cover	min_meas	num_part	err. rate↓
$entropy()$	1	0.00	1.25	17.49%
$ORT()$	1	0.00	1.25	17.79%
$DI_n()$	1	0.00	1.50	18.23%
$entropy_+()$	1	0.00	4.00	18.97%
$ORT_+()$	1	0.00	1.50	19.87%
$DI_{n+}()$	1	0.00	1.25	20.33%

A comparison with the optimized parameter combinations in Table 7.2.2 shows that no

Table 7.7: The pruned version of the $ORT_{+}()$ function, even though it achieved the lowest accuracy of the all the pruned versions, is more accurate on the 5 datasets than the unpruned version of the algorithm which achieved the highest accuracy (*entropy*()).

Algorithm	lower on	avg err. rate (%)	clear wins
$ORT_{+}()$ w/pruning	**3**	**16.72**	**2**
entropy() w/out pruning	2	17.49	1
Subtract	1	-0.7	1

unpruned version of the algorithm achieved a lower error rate than any of the pruned versions. Further analysis showed that all pruned version of the algorithm were more accurate than all the unpruned versions. Table 7.7 presents the results of the two versions that came closest to contradicting this.

This evidence supports a claim that pruning improves DBPredictor's accuracy. Further evidence to this effect is presented in Section 7.3.5. Part of this section investigates the effect of pruning on overfitting.

Observation: Another observation from Table 7.6 is that all parent-child variations led to a higher error rate than all of the sibling-sibling versions of F(). The question of which method to calculate F() is investigated in the next section.

7.2.4 Sibling-Sibling versus Parent-Child

Because the three functions based on the *parent-child* variation achieved a higher error rate than their *sibling-sibling* counterparts in both Table 7.6 and Table 7.7, it appears that the use of this approach may be rejected. Indeed, after further analysis, all the algorithms (which made use of pruning) based on the *sibling-sibling* approach to the heuristic function calculation, were found to be more accurate than all *parent-child* based functions on the 5 tested datasets. Table 7.8 presents the two versions that came closest to contradicting this statement.

Based on this consistent response, there is strong evidence that the sibling-sibling approach achieves a lower error rate than the parent-child approach.

Table 7.8: Comparison of the algorithm with the least accurate sibling-sibling heuristic function and the most accurate parent-child variation of the heuristic functions. Pruning is turned on.

f()	lower on	avg	clear wins
ORT()	3	15.30%	2
DI_{n+}()	2	15.33%	0
Subtract	1	0.03%	2

Table 7.9: Summary of accuracy results against the 5 datasets for the *DI*() and *entropy*() versions of DBPredictor.

Algorithm	lower on	avg err. rate (%)	clear wins
DI()	**3**	**15.13**	0
entropy()	2	15.18	0
Subtraction	1	-0.05	0

7.2.5 Selection of Accurate F()

Now that the pruning has been validated and the parent-child variation has been rejected, this section determines which measure to pass on to the remaining investigations. Unfortunately, based on the five datasets that were set aside, no version emerged as a clear winner. As can be recalled from table 7.2.2, the DI_n() based algorithm achieved the lowest average error rate. Upon further analysis, this version of the algorithm also achieves a lower error rate on more datasets than both the *entropy*() and the *ORT*() based algorithms. However, no algorithm was able to achieve a 99.5% confidence win on any of the five datasets. Table 7.9, compares the DI_n() based algorithm, to the next most accurate version: *entropy*().

Because the DI_n() based algorithm met two of the three criteria and because no other algorithm achieve a lower error rate with 99.5% confidence, this version of the DBPredictor was passed to the remaining phases of the empirical study.

Observation: One surprising discovery from this investigation is the effectiveness of the *DIn*() evaluation function which has not been used extensively in previous examinations of classification algorithms. This function measures the Euclidean distance between two class

probability distribution vectors. One attraction to this measure is that because it is easily visualized, more people may be able to understand the functioning of the classifier.

7.3 Relative Accuracy of DBPredictor

The main task of the study reported in the previous section was to locate a relatively accurate version of DBPredictor. The main task for the study covered in this section is to determine the suitability of this version of DBPredictor, for on-line classification tasks. Because of the prevalence of irrelevant attributes and underspecified events in on-line classification tasks, the question of suitability is investigated with the following questions:

- How does DBPredictor's accuracy compare to that of C4.5 and IB1 on general datasets?
- How does DBPredictor's accuracy compare to that of C4.5 and IB1 when datasets are known to contain several irrelevant attributes?
- How does DBPredictor's accuracy compare to that of C4.5 and IB1 when event vectors are known to contain several missing attribute-values?

Two secondary questions are also investigated in order to better understand DBPredictor's behaviour:

- Relative to C4.5 and IB1, does DBPredictor overspecialize?
- Relative to C4.5 and IB1, is DBPredictor's biased for or against datasets with numerical attributes?

After a brief review of the benchmark algorithms in Section 7.3.1, the results for each of these five questions are presented in the same order as above.

7.3.1 Benchmark Algorithms

Three benchmark algorithms were selected to identify DBPredictor's relative accuracy: the naive, C4.5 and IB1 classification algorithms.

Naive Classifier

The simplest classification algorithm used in the empirical study returned the most common class of the dataset. This classifier is sometimes referred to as the *naive classifier*. The performance of this algorithm on the 23 datasets can be derived from the class distribution column in Table 7.1. For the *echocardiogram* dataset (EC), for example, there are 132 records and three classes. It's most common class however occupies 81.8% of its records (i.e. 108 records). When only this information is available, the best strategy of a naive classification algorithm is to always assign the most populous class. In the case of the *echocardiogram* dataset the naive classifier achieves an error rate of (100%-81.8%) 18.2%.

C4.5r8

The representative eager model-based classification algorithm used in the empirical study was the C4.5 program [58]. C4.5 is a state-of-the-art classification algorithm that constructs decision trees that can then be used to classify new cases. For this study, release 8 was used, which improves C4.5's performance on datasets with continuous attributes [60].

IB1

The representative lazy instance-based classification algorithm used in the empirical study was the IB1 algorithm [6]. This algorithm is presented in Section 3.1. The algorithm contains one threshold parameter, k, which determines the size of the instance set to base each prediction on. To set this parameter, IB1 was optimized on the same five datasets as DBPredictor. Based on this investigation, k was set to 20.

7.3.2 General Comparison of Accuracy

The first question addressed in this study was how DBPredictor's general accuracy performance faired against the benchmark algorithms. The accuracy results for each of the algorithms on the 23 datasets are reported in Table 7.10. A quick scan through this table does not show a clear winner. C4.5 achieves the lowest average error rate, IB1 was most accurate on more datasets and DBPredictor tied for the average ordinal. Further analysis, showed that aside for the naive classifier, no algorithm was more accurate than another. While C4.5 achieved a lower average error rate than DBPredictor, C4.5 did not achieve a lower error rate than DBPredictor on more datasets (12 versus 10). Similarly, though in

opposite order, while IB1 did achieve a lower error rate on more datasets than DBPredictor (12 versus 10), IB1 did not achieve a lower average error rate than DBPredictor. These results present evidence that DBPredictor has attained parity with C4.5 and IB1 in terms of general accuracy.

7.3.3 Irrelevant Attributes

The next experiment investigated DBPredictor's relative accuracy with respect to the presence of irrelevant attributes. To perform this test, ten attributes were added to each of the datasets. Each dataset's proportion of numerical to symbolic attributes was maintained. The values for each of these attributes were selected randomly. Two value distributions were tested: even and Gaussian. Because the Gaussian distribution produced slightly larger error rates, it was selected for this test.

The results for this experiment are summarized in table 7.11. As expected, IB1's accuracy is relatively sensitive to irrelevant attributes. DBPredictor's response, on the other hand, is closely related to the accuracy of the more robust C4.5 algorithm. When formally compared, DBPredictor and C4.5 were found to be more accurate than IB1 when tested against the updated datasets. On the other hand, neither DBPredictor nor C4.5, were more accurate than each other on these same datasets.

These results present evidence that DBPredictor's accuracy does not degenerate as fast as lazy instance-based algorithms in the presence of irrelevant attributes. Also, DBPredictor remains equivalent in accuracy performance to eager model-based algorithms in the presence of irrelevant attributes.

7.3.4 Underspecified Event Vectors

All the previous tests have assumed that all the information about each event vector was known when making the prediction request. The next experiment that was performed, investigated DBPredictor's relative accuracy with respect to the presence of underspecified event vectors. To perform this test different proportions of the event vector were obscured from the classification request. The selection of attributes to hide occurred randomly for

Table 7.10: Accuracy results on the 23 datasets for DBPredictor and the three benchmark algorithms. For each dataset the placement ordinal [1-4], average error rate (%) and standard deviation, is given for each algorithm

#	abbrv.	naive σ 0.0	DBP	C4.5	IB1
1	AN	4 23.80	2 7.70σ0.41	1 7.46σ0.51	3 12.18σ0.33
2	AD	4 74.78	2 34.38σ0.69	1 20.95σ1.04	3 44.02σ0.82
3	BW	4 34.50	2 3.73σ0.24	3 5.57σ0.51	1 3.67σ0.05
4	CH	4 47.80	2 1.27σ0.05	1 0.58σ0.09	3 5.75σ0.30
5	CA	4 44.50	3 15.12σ0.43	2 14.68σ0.59	1 14.18σ0.19
6	CG	4 30.00	2 28.01σ1.13	3 28.34σ0.95	1 26.60σ0.47
7	DI	4 34.90	1 24.85σ0.83	2 25.75σ1.12	3 26.16σ0.43
8	EC	2 18.20	4 23.52σ1.23	3 18.82σ1.11	1 18.02σ0.07
9	GL	4 64.48	2 32.14σ2.05	1 31.71σ2.10	3 37.50σ1.37
10	HR	4 59.37	1 16.04σ1.09	2 25.41σ2.12	3 33.30σ2.33
11	HT	4 44.40	2 21.45σ1.53	3 22.05σ1.80	1 18.07σ0.39
12	HC	3 45.90	2 45.88σ0.98	4 47.96σ1.72	1 44.02σ1.55
13	HH	4 36.10	3 22.38σ0.88	2 20.63σ1.10	1 18.33σ1.13
14	HE	3 20.60	2 19.88σ1.44	4 20.81σ1.64	1 16.68σ0.56
15	HO	4 37.00	3 20.88σ0.82	1 15.81σ0.77	2 17.77σ0.35
16	IR	4 37.00	1 4.34σ0.80	3 5.25σ0.89	2 4.40σ0.37
17	LT	4 95.90	2 9.00σ0.46	3 11.99σ0.32	1 6.50σ0.27
18	LD	4 42.03	3 36.90σ2.46	1 33.24σ1.72	2 36.70σ2.08
19	MU	4 48.20	1 0.00σ0.00	1 0.00σ0.00	3 0.10σ0.00
20	SE	4 85.71	2 5.18σ0.27	1 3.31σ0.26	3 5.83σ0.05
21	SO	4 36.17	1 2.12σ0.25	2 2.87σ1.25	3 17.88σ2.29
22	TO	4 34.70	2 10.33σ1.10	3 14.44σ0.90	1 1.79σ0.50
23	VO	4 38.60	1 4.60σ0.61	2 4.82σ0.40	3 8.05σ0.42
Average		3.8 44.99	2.0 16.94σ0.86	2.1 16.63σ1.00	2.0 18.16σ0.67

Table 7.11: Average error rate for DBPredictor, C4.5 and IB1 when 0 and 10 irrelevant attributes were added to each of the datasets.

Irrelev.	DBP	C4.5	IB1
0	16.94	16.63	18.16
10	18.61	17.96	23.80
Δ	9.9%	8.0%	31.1%

Table 7.12: Summary of error rate results when 0%, 25%, 50% and 75% of an event vector's attributes were uninstantiated.

Missing %	DBP	C4.5	IB1	naive
0%	16.94	16.63	18.16	44.99
25%	19.87	21.65	21.16	44.99
50%	21.25	26.13	24.30	44.99
75%	28.34	35.45	28.49	44.99

each prediction. When the proportion was set to 50%, for example, half of the event vector's attributes were randomly selected and their values were set to unknown.

Three proportions of uninstantiated vectors were tested: 25%, 50% and 75%. The results of these tests are summarized in table 7.12. A quick scan through this table shows that C4.5's accuracy degrades more rapidly than DBPredictor's or IB1's. Further analysis showed that C4.5 was indeed less accurate than DBPredictor and IB1 on all three proportions of underspecified event vectors. A comparison between DBPredictor and IB1 showed that neither algorithm was more accurate than the other for any of the tested proportions.

These results present evidence that in settings were a significant proportion of an event vector's attributes will be uninstantiated, the DBPredictor algorithm will likely produce more accurate classifications than the eager model-based C4.5 classification algorithm.

7.3.5 Overspecialization

Another measure of interest for a classification algorithm is its tendency to overspecialize (overfit). This effect is commonly tested by comparing the accuracy of the naive classifier against another classifier. This study determined the tendency of DBPredictor, C4.5 and IB1 to overspecialize to determine DBPredictor's relative vulnerability at overspecialization. Also within this study the unpruned DBPredictor was tested against the naive classifier to further understand the impact of pruning on DBPredictor's accuracy.

The naive classifier is commonly used to determine another classification algorithm's tendency at overspecialization. The effect of a classification algorithm performing worse than

Table 7.13: Datasets in which DBPredictor, C4.5 and IB1 were not more accurate than the naive classifier with more than 99.5% confidence. The second column Δ presents the difference between the two error rates.

DBP		C4.5		IB1	
D	Δ	*D*	Δ	*D*	Δ
EC	-5.32	HC	-1.77	EC	0.18
HC	0.02	EC	-0.62	HC	1.93
HE	0.72	HE	-0.33		

the naive classifier can be understood through a worst case example were the algorithm in question overfits each prediction such that it is always supported by only a single record [13]. Suppose that there is a 70%/30% division among a binary class attribute that is to be predicted by a set of random attributes (i.e. 100% noise). Each rule will be correct 70% of the time on the training set and therefore be on par with the naive classifier. When applied to new data however, the rules that predicted the majority class will now only be accurate a further 70% of the time. Therefore, the algorithm will only be correct $70\% \times 70\% = 49\%$ of the time.

A quick scan through Table 7.10 shows that DBPredictor performed worse than the naive classifier on one dataset, C4.5 on three, and IB1 on none. Table 7.13 presents the datasets that each algorithm was unable to achieve a lower error rate than the naive classifier with a 99.5% confidence.

Based on these results there is evidence that DBPredictor is prone to overspecialization. However, the algorithm is not significantly more prone to overspecialization than C4.5.

Overspecialization Without Pruning

To extend our understanding of the impact of pruning on DBPredictor's accuracy, a test was performed to determine if the lack of pruning had a significant impact on overspecialization. When the accuracy of the most accurate unpruned version of DBPredictor was compared to the naive classifier's performance on the 23 datasets, DBPredictor achieved a higher error rate on five datasets: *liver-disease*, *hepatitis*, *heart-c*, *credit-g*, and *echocardiogram*. Based on this evidence pruning appears to significantly lower DBPredictor's vulnerability to overspecialization.

Table 7.14: Average percentage of numerical attributes in the datasets that each algorithm performed significantly more accurately on, when compared to one other algorithm. When compared to IB1, DBPredictor contained on average 39% numerical attributes in the datasets that it performed more accurately on. When compared to IB1's proportion (52%), the difference between the two Δ, is equal to $\frac{39}{52} \approx 75\%$.

tested pair	DBP/C4.5	DBP/IB1	C4.5/IB1
proportions	54%/55%	39%/52%	20%/57%
Δ	98%	75%	35%

7.3.6 Numerical Domains

The final empirical study investigated whether DBPredictor's handling of numerical attributes biased its accuracy with respect to the proportion of numerical attributes in a dataset. For each pairing of DBPredictor, C4.5 and IB1, the datasets in which one algorithm performed significantly more accurately in[2] than the other were identified. For example, DBPredictor performed significantly more accurately than IB1 on nine datasets, while IB1 performed significantly more accurately than DBPredictor on seven datasets. Next, the average proportion of numerical attributes on these two groups of datasets was evaluated. For example, approximately 39% of the attributes for the nine datasets which DBPredictor performed significantly better on, were numerical. Because the average number of numerical attribute among the 23 datasets is 48%, it is expected that an unbiased algorithm will also achieve this proportion of numerical attribute among its more and less accurate datasets. Table 7.14 summarizes the results of the three pairwise tests.

When tested against each other, C4.5 and DBPredictor performed similarly. Both had a slight bias for numerical attributes. When compared to IB1 however, DBPredictor appears to be significantly biased against numerical datasets. However, because of C4.5's significant bias against numerical attributes relative to IB1, and because of the well known strength of k-NN based algorithms in numerical domain's [66], DBPredictor's approach to numerical attributes appears to be sound.

[2]based on a 95% confidence one-tailed t-test

7.4 Discussion

The focus of this chapter was to determine DBPredictor's suitability for on-line classification tasks with respect to accuracy. To test its suitability, DBPredictor's accuracy was compared to the performance of C4.5 and IB1 against general datasets, datasets with irrelevant attributes and underspecified event vectors. These three tests achieved the following results:

- When tested against general datasets no algorithm was more accurate than the other.
- When tested against databases with irrelevant attributes, DBPredictor and C4.5 were more accurate than IB1.
- When tested against event vectors with missing attribute-values DBPredictor and IB1 were more accurate than C4.5

Given this response, DBPredictor appears to be the most suitable choice for an on-line classification task, with respect to accuracy.

Because of DBPredictor's positive results, a few other questions were investigated to better understand its underlying behaviour.

- Pruning significantly reduces DBPredictor's vulnerability to overspecializes and also improves accuracy in general.
- Even with pruning, DBPredictor overspecializes. However this response appears to be no worse than C4.5's response.

The use of pruning has been strongly validated, however, based on IB1's ability to avoid overspecialization, there appears to be room for improvement. A more formal investigation of pruning may be helpful.

- The parent-child calculation for the heuristic function significantly degrades DBPredictor's accuracy.

The evidence is quite strong that the parent-child approach should be rejected.

- The $DI_n()$ heuristic function, resulted in the lowest average error rate, but it was not clearly superior than the *entropy*() based function.

Because of *entropy*()'s popularity and $DI()$'s relative obscurity, this result was unexpected. This finding may be useful for other top-down induction algorithms. Another possible advantage to the use of this function is the simplicity of its visualization, and therefore possibly greater acceptance.

- DBPredictor's handling of numerical attributes does not appear to significantly bias the algorithm's relative accuracy for or against datasets with numerical attributes.

The enhanced approach to numerical attributes handling was defined to make the use of DBPredictor more convenient. This positive result, however, supports further investigation into this method.

7.5 Chapter Summary

This chapter presented the results of an empirical study into DBPredictor's accuracy. The main focus of the investigation was to test DBPredictor's suitability for on-line classification tasks with respect to accuracy. Several other secondary questions were investigated to develop a better understanding of DBPredictor's accuracy in general. The study followed the common approach of locating a group of representative datasets, selecting an error rate estimation method and defining a set of criteria to tests hypothesis on an algorithm's accuracy. The first set of tests attempted to discover an accurate combination of parameter settings and heuristic function for DBPredictor. In the process of locating this combination, a few other secondary questions into pruning, and the child-parent approach were investigated. Next, DBPredictor was compared to the C4.5 and IB1 classification algorithms. Finally, the impact of the study's results were discussed.

Chapter 8

Empirical Study of Running Time

While a classification algorithm may be very accurate, the amount of time that it expends before it returns its prediction, is another critical measure of the algorithm's success. This chapter presents the results of a brief empirical study of DBPredictor's running time performance. The main question addressed within this study was DBPredictor's suitability for on-line classification tasks, with respect to running time performance. In summary, based on the performance on one large dataset, IB1 solved tasks 4 to 8 times faster than DBPredictor, which in turn solved tasks 50 to 100 times faster than C4.5. This result shows that a tradeoff is likely between time, and the ability to produce rule based results with robust accuracy.

This chapter is organized into three sections. Section 8.1 presents the methodology used to validate empirical statements of running time. Section 8.2 reports the study used to achieve a general understanding of DBPredictor's real-world response to different numbers of attributes and records. Finally, Section 8.3 reports on an initial investigation into DBPredictor's relative performance relative to the C4.5 and IB1 classification algorithms.

8.1 Methodology

This section describes the methodology used to test a classification algorithm's empirical running time. Unfortunately, empirical running time performance evaluations are not

commonly reported in the classification literature. To test an algorithm's running time performance, this study performed experiments against different proportions of records and attributes from the same domain. To achieve this, a dataset was located which remained sizeable, even when only a small portion of its records and attributes were present. For this study, the *census-year* dataset was used. This dataset is composed of m=199,523 records and $m-1$=37 predicting attributes (13 of these being numerical). The class attribute of this dataset has c=2 classes of relatively equal proportion and some missing attribute-values. This dataset is located at the UCI repository [51].

Several decisions were also made about the testing environment. Because the C4.5 and IB1 program are not implemented to communicate with an RDBMS, all tests were performed in memory. Furthermore, since C4.5 makes use of significant space resources to achieve its classification, DBPredictor made use of its time efficient search technique. This technique was found to consume an equivalent amount of memory space to achieve its prediction. All experiments in this section were performed on a dedicated computer with the following configuration: 133Mhz Pentium CPU, 64MB DRAM, and the Linux 2.0.31 O/S. All classification algorithms were also implemented with the same (GNU) ANSI-C compiler and optimization flags.

8.2 Standalone Performance

To investigate DBPredictor's real-world response to different sizes of n and m, the *census-year* dataset was broken-up in both directions and the average time required for each classification request was recorded. To test the response in the n dimension, four proportions of the dataset were tested: 100%, 75%, 50% and 25% of the records (ie. 199,523, 149,642, 99,762 and 49,881 records). The records for the smaller datasets were randomly selected while keeping a similar distribution of the values in the class attribute. Similarly, to test the response in the m dimension, tests were performed by obscuring different numbers of attributes: 0, 4, 8, 12, 16, 20, 24, 28, 32, and 36. To calculate an average completion time per request, one hundred records were randomly selected from the accompanying testset to the *census-year* dataset. The results of this study are located in Table 8.2.

Time performance results (in seconds) for DBPredictor against different proportions of the *census-year* dataset. The dataset has 37 predicting attributes and $\approx 200,000$ records.

Instantiated Attributes	Records			
	25%	50%	75%	100%
1 (2.7%)	0.35	0.94	1.67	2.14
5 (13.5%)	0.74	1.73	3.58	6.23
9 (24.3%)	1.24	2.74	5.44	8.97
13 (35.1%)	1.97	4.34	8.95	12.70
17 (45.9%)	2.83	6.59	10.70	16.64
21 (56.8%)	3.79	8.79	14.07	20.22
25 (67.6%)	4.89	11.66	17.41	25.79
29 (78.4%)	6.25	14.37	20.72	30.30
33 (89.2%)	7.42	17.12	24.54	34.03
37 (100%)	8.90	20.74	28.01	38.22

On average, when 100% of the records were present and when 100% of the event vector was instantiated DBPredictor required 38.22 seconds to report its prediction. When only half as many records were used the average time is cut to 20.74 seconds. Generally the algorithm's time performance appears to grow linearly with the number of records in the dataset. In the m dimension, when only half of the attributes were instantiated and all the records were present, the algorithm required approximately 18 seconds to report its result. Generally the algorithm's time performance appears to grow slightly faster than linearly with respect to the number of attributes.

8.3 Relative Performance

The final question investigated in the empirical study was the difference in time response between DBPredictor and the IB1 and C4.5 algorithms on different proportions of the *census-year* dataset. The performance difference between all three algorithm was significant. IB1 was 4 to 8 times faster than DBPredictor, which in turn was 50 to 100 times faster than C4.5. Because of this, the comparisons below, report how many classification requests the faster algorithm could achieve before the slower algorithm had classified a single event.

Table 8.1: Number of classification performed by DBPredictor before C4.5 returns its first classification. Different proportions of the *census-year* dataset were tested.

Instantiated Attributes	Records			
	25%	50%	75%	100%
9 (24.3%)	46	51	66	72
18 (48.6%)	52	68	86	103
28 (75.7%)	48	61	93	109
37 (100%)	42	53	85	104

As in Section 8.2, different settings of n and m were investigated. For n, four settings were investigated: 100%, 75%, 50% and 25%. For m, four settings were investigated: 100%, 75.7% (28/37) 48.6% (18/37), and 24.3% (9/37). Because of the significant amount of time required by C4.5 to build its decision trees, only a single attribute combination was selected for the three settings in which attributes were obscured. Care was taken to ensure that the proportion of numeric and symbolic attributes remained the same (13/25).

Comparison with C4.5: Table 8.1 presents DBPredictor's relative running time when compared to C4.5's performance. On average, when all of the records were present and when the event vectors were fully instantiated, DBPredictor was able to complete 104 predictions against the *census-year* dataset, before C4.5 completed a single prediction. When the number of records was halved ($n \approx 100,000$ records), DBPredictor was able to make approximately one half as many predictions (53). If, instead, the number of attributes was pruned by approximately one half (m=18), DBPredictor continued to make an equal number of predictions (103). These results suggest that DBPredictor solves on-line classification significantly faster than C4.5. Furthermore, this gap appears to grow linearly as the number of records (n) grows, and remain steady as the number of attributes (m) grows.

Table 8.2: Number of classification performed by IB1 before DBPredictor returns its first classification. Different proportions of the *census-year* dataset were tested.

Instantiated Attributes	Records			
	25%	50%	75%	100%
9 (24.3%)	4.0	4.7	5.0	5.9
18 (48.6%)	4.3	5.3	6.1	6.8
28 (75.7%)	6.4	6.9	7.1	7.4
37 (100%)	7.4	7.6	7.7	7.9

Comparison with IB1: Table 8.2 presents DBPredictor's relative running time when compared to IB1's performance. On average, when all of the records were present and when the event vectors were fully instantiated, IB1 was able to complete approximately 7.9 predictions against the *census-year* dataset, before DBPredictor completed a single prediction. When the number of records was halved ($n \approx 100,000$ records), IB1 was able to make a relatively similar number of predictions.. If, instead, the number of attributes was pruned by approximately one half (m=18), IB1 made fewer predictions (6.8). These results indicate that IB1 solves on-line classification faster than DBPredictor. This gap, appears to grow linearly with the number of attributes (m), and remain steady with the number of records (n).

8.4 Chapter Summary and Discussion

This chapter presented the results of a brief empirical investigation into DBPredictor's time performance. The large *census-year* dataset was selected to facilitate this portion of the study. Again, the C4.5 and IB1 algorithms served as the benchmark algorithms. The first study showed that the time efficient version of the algorithm may, on average, grow linearly with then number of records and grow larger than linearly with the number of attributes. When compared to C4.5, DBPredictor showed a significant time performance gain. This gain grew approximately as a linear function of the number of attributes in the dataset. When compared to IB1, however, DBPredictor attained a slower running time performance. This gap grew approximately, as a liner function of the number of attributes.

These results indicate that, when compared to IB1, the benefits of DBPredictor's rule based result and robust accuracy comes at the expense of increased running time.

Chapter 9

Conclusion

In this thesis, we described a framework for knowledge based on-line classification tasks and proposed an algorithm, named DBPredictor, that is particularly suited to these tasks. This final chapter summarizes the approach taken by the thesis, reviews the key contributions to the field of classification, and speculates on future research directions.

Section 9.1 presents the thesis summary. Next, Section 9.2 summarizes the key contributions of the thesis. In Section 9.3, we suggest some possible areas for future research. And finally, Section 9.4 presents some concluding remarks.

9.1 Thesis Summary

Chapter 2 began the thesis by describing a framework for knowledge based on-line classification tasks. These tasks provide a database, event vector and class attribute and require a class prediction that should be justified with the use of a high-level representation. Optional constraints on how the algorithm achieves its task, includes the requirement of direct communication against a database. As in most classification tasks, the candidate algorithm will be measured by its accuracy, speed and resource requirements. The understandability of its prediction and its operation, while subjective, are also important to the task. Finally, some areas related to on-line classification, but out of scope for this thesis, include regression, batch classification and system guided classification.

Chapter 3 presented a survey of classification algorithms that may be used for on-line classification. Historically, Machine Learning classification algorithms have made use of either a lazy instance-based approach, or an eager model-based approach. As specific

instances of these approaches, we presented the k nearest-neighbour based IB1 algorithm, and the use of top down induction of decision trees. The third approach presented made use of lazy classification with dynamic relevance analysis. This approach appears more in tune with the requirements of on-line classification. Several algorithms have been recently proposed that make use of this approach. The local induction of decision tree algorithm, for example, combines the lazy instance-based approach and top-down induction of decision trees into a hybrid approach. The other, more intensively surveyed approach made use of lazy model-based induction. This is the approach used by DBPredictor, however the LazyDT algorithm is reviewed instead, to highlight the differences in implementation between the two algorithms.

Chapter 4 described the greedy top-down heuristic search used by DBPredictor to locate a classification rule. The description included the natural support for numerical and concept hierarchy attributes, as well as the use of a tightly-coupled interface to an SQL database. The chapter concluded with an analysis of the algorithm's complexity. This proved that the algorithm's running time complexity is $O(nm^3h)$ and its space complexity is $O(m^2)$, where n is the number of records, m is the number of attributes, and h is the maximum level of specialization levels for hierarchical or numerical attributes.

Chapter 5 presented an alternate search technique for DBPredictor that is more in-line with the space assumptions of current machine learning algorithm implementations. While this approach achieves a lower running time complexity of $O(nm^2h)$ than the search technique described in the previous chapter, its space complexity is increases to $O(nm+m^2)$. This version of the algorithm is therefore referred to as the time efficient version while the previous version is referred to as the space efficient version.

Chapter 6 considered several versions of the heuristic function used by DBPredictor to navigate its rule space. The first version category, considered three different base measures ($entropy()$, $ORT()$, $DI_n()$) and also two methods of calculation (sibling-sibling and parent-child). A simple pruning technique was also proposed to determine DBPredictor's response to event a simple pruning mechanism.

Chapter 7 reported the results of the empirical investigations into DBPredictor's accuracy. Initially, the different versions of the heuristic function were tested to locate a generally accurate version of DBPredictor. The second portion of the study investigated DBPredictor's relative suitability for on-line classification tasks. DBPredictor was found to be as accurate in general as IB1 and C4.5, but more accurate than IB1 in the presence of

irrelevant attributes, and more accurate than C4.5 in the presence of underspecified event vectors.

Chapter 8 concluded the empirical study of DBPredictor's performance with a brief investigation of the algorithm's real-world running time. With the use of a large dataset, DBPredictor' raw and relative running time was tested against different proportions of the dataset. When compared to C4.5, DBPredictor was able to satisfy a significant number of on-line classification requests before C4.5 could classify a single event. DBPredictor, however was slower than IB1, although not by as significant a factor as the difference between DBPredictor and C4.5.

9.2 Contributions

The main contribution of this thesis is a lazy model-based algorithm, named DBPredictor, that is particularly suited to knowledge based on-line classification tasks.

When compared to eager model-based approaches, such as C4.5, and lazy instance-based approaches, such as IB1, DBPredictor achieves an alternate balance of accuracy and running time. With respect to accuracy, DBPredictor is more robust to the presence of irrelevant attributes and the underspecified event vectors. With respect to running time, DBPredictor is shown to be more effective than C4.5, but less effective than IB1.

Contributions in this thesis towards lazy model-based classification in general include:

- Dynamic handling of numerical attributes. This approach allows a lazy model-based classification algorithm to avoid the inconvenience and cost of global discretization. Empirical results show that the bias of this approach against numerical attributes is less severe than the bias of the C4.5r8 classifier.
- the rejection of the parent-child calculation approach.
- support for tightly-coupled database connections and for concept hierarchies.

9.3 Future Research

This section indulges in some speculation about future research directions for on-line classification tasks and in particular, the use of lazy model-based induction for these tasks. Some suggestions are in the form of refinements, others are more speculative.

- One possible area of further research is in a formalized approach to pruning. Now that the value of pruning has been explicitly shown with the `min_cover`, `min_meas` thresholds, a more formal approach may prove even more effective. One possibility that comes to mind is the chi-square test for stochastic independence proposed in [57].

- Another area of interest is to extend DBPredictor's representation language to support negation within propositions that refer to symbolic attributes. In this way, DBPredictor may achieve more accurate models. This extension to DBPredictor is likely related to the approach used by the LazyDT algorithm. If, as for LazyDT, this extension results in a significant number of ties, then the use of limited lookahead may prove fruitful.

- DBPredictor's general accuracy in the presence of records with missing attribute-values needs to be investigated. While DBPredictor is likely sensitive to this situation, no lazy technique to this shortcoming is apparent. One possible technique around this problem may be to retreive the most similar records to the event vector (see next suggestion) and determine the missing attribute-values just for these records.

- A more speculative extension to DBPredictor is to base the `seed_rule()` procedure on a high-level instance-based search. In this way, a significant amount of effort may be saved by quickly zooming in on the region of interest. The algorithm's accuracy may also benefit from the same effect encountered by the proposal for local induction of decision trees [30]. The challenge to this investigation would likely be the generation of a simple rule that covers these "similar" records.

- Finally, a more ambitious direction is in the support of interactive classification. Currently LazyDT and DBPredictor are applicable to data driven tasks in which a partially instantiated unlabeled event is presented. Some domains will likely benefit from an algorithm that can suggests which attributes to also instantiate for the event in question.

9.4 Concluding Remarks

Classification is essential to all life. Because more and more of our observations are being stored in databases, the value of classification based on these structured repositories will also increase. To tap into this opportunity, this thesis proposes an algorithm named DBPredictor for the task of knowledge based on-line classification. Rather than eagerly developing a model to support every conceivable classification request, the approach of this algorithm is to wait for each classification request to appear and then use of a lazy model-based approach to return an accurate and understandable prediction. The preliminary empirical investigations presented in this thesis, indicates that DBPredictor is a strong candidate for knowledge based on-line classification tasks.

Bibliography

[1] AAAI. *Thirteenth National Conference on Artificial Intelligence.* AAAI Press, 1996.

[2] R. Agrawal and J. C. Shafer. Parallel mining of association rules: Design, implementation, and experience. *IEEE Trans. Knowledge and Data Engineering*, 8:962–969, 1996.

[3] D. W. Aha. *A Study of Instance-Based Algorithms for Supervised Learning Tasks.* PhD thesis, University of California, Irvine, 1990.

[4] D. W. Aha, editor. *Lazy Learning.* Kluwer Academic, May 1997.

[5] D. W. Aha. Lazy learning editorial remarks. [4], pages 1–3.

[6] D. W. Aha, D. Kibler, and M. K. Albert. Instance-based learning algorithms. *Machine Learning*, 6(1):37–66, 1991.

[7] C. Apte and S. J. Hong. Predicting equity returns from securities data with minimal rule generation.

[8] M. A. Arbib, A. J. Kfoury, and R. N. Moll. *A basis for theoretical computer science.* Texts and monographs in computer science. Springer-Verlag, 1981.

[9] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Wadsworth, 1984.

[10] J. G. Carbonell, R. S. Michalski, and T. M. Mitchell. An overview of machine learning. In Michalski et al., editor, *Machine Learning: An Artificial Intelligence Approach, Vol. 1*, pages 3–24. Morgan Kaufmann, 1983.

[11] J. Catlett. *Megainduction: Machine Learning on Very Large Databases.* PhD thesis, University of Sydney, June 1991.

[12] J. Cheng, U. M. Fayyad, K. B. Irani, and Z. Qian. Improved decision trees: a generalized version of id3. In *Proc. Fifth Int. Conf. Machine Learning*, pages 100–107, San Mateo, California, 1988.

[13] P. Clark and R. Boswell. Rule induction with CN2: Some recent improvements. In *Machine Learning - Proceedings of the Fifth European Conference (EWSL-91)*, pages 151–163. Springer-Verlag, 1991.

[14] P. Clark and T. Niblett. The CN2 induction algorithm. *Machine Learning*, 3:261–283, 1989.

[15] E. F Codd, S. B. Codd, and C. T. Salley. Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate. In *E. F. Codd & Associates available at http://www.arborsoft.com/OLAP.html*, 1993.

[16] S. Cost and S. Salzberg. A weighted nearest neighbor algorithm for learning with symbolic features. *Machine Learning*, 10:57–78, 1993.

[17] T. M. Cover and P. E. Hart. Nearest neighbor pattern classification. *IEEE Transaction on Information Theory*, 13:21–27, 1967.

[18] T. Dietterich. Overfitting and undercomputing in machine learning. *ACM Computing Surveys*, 27(3):326–327, September 1995.

[19] T. G. Dietterich and R. S. Michalski. A comparative review of selected methods for learning from examples. In Michalski et al., editor, *Machine Learning: An Artificial Intelligence Approach, Vol. 1*, pages 41–82. Morgan Kaufmann, 1983.

[20] A. J. Dobson. *An Introduction to Generalized Linear Models.* Chapman & Hall, 1990.

[21] P. Domingos. Unifying instance-based and rule-based induction. *Machine Learning*, 24(2):141–168, August 1996.

[22] P. Domingos. Context-sensitive feature selection for lazy learners. In Aha [4], pages 227–253.

[23] J. Dougherty, R. Kohavi, and M. Sahami. Supervised and unsupervised discretization of continous features. In A. Prieditis and S. Russel, editors, *Machine Learning: Proceedings of the Twelfth International Conference*, pages 194–202. Morgan Kaufmann.

[24] U. M. Fayyad, S. G. Djorgovski, and N. Weir. Automating the analysis and cataloging of sky surveys. In U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 471–493. AAAI/MIT Press, 1996.

[25] U. M. Fayyad and K. B. Irani. On the handling of continuous-value attributes in decision tree generation. *Machine Learning*, pages 87–102.

[26] U. M. Fayyad and K. B. Irani. The attribute selection problem in decision tree generation. In *Proceedings of the Tenth National Conference of Artificial Intelligence*, pages 104–110. AAAI Press, 1992.

[27] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996.

[28] E. Fix and Jr. J. L. Hodges. Discriminatory analysis, nonparametric discrimination, consistency properties. Technical Report 4, United States Air Force, School of Aviation Medicine, Randolph Field, TX, 1951.

[29] J. H. Friedman, R. Kohavi, and Y. Yun. Lazy decision trees. [1], pages 717–724.

[30] T. Fulton, S. Kasif, S. Salzberg, and D. Waltz. Local induction of decision trees: Towards interactive data mining. In Simoudis et al. [63], pages 14–19.

[31] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab and sub-totals. *Data Mining and Knowledge Discovery*, 1:29–54, 1997.

[32] R. Greiner, A. J. Grove, and D. Roth. Learning active classifiers. In Lorenza Saitta, editor, *Proc. of 13th Int. Conf. Machine Learning (ICML'96)*, pages 207–215. Morgan Kaufmann, 1996.

[33] J. Han, Y. Cai, and N. Cercone. Knowledge discovery in databases: An attribute-oriented approach. In *Proc. 18th Int. Conf. Very Large Data Bases*, pages 547–559, Vancouver, Canada, August 1992.

[34] R. V. Hogg and J. Ledolter. *Engineering Statistics.* Macmillan Publishing Company, 1987.

[35] X. Hu. Conceptual clustering and concept hierarchies in knowledge discovery. Master's thesis, Simon Fraser University, School of Computing Science, December 1992.

[36] P. J. Huber. From large to huge: A statistician's reactions to KDD & DM. In *The Third International Conference on Knowledge Discovery & Data Mining*, pages 304–308, 1997.

[37] E. B. Hunt, J. Marin, and P. T. Stone. *Experiments in Induction.* Academic Press, 1966.

[38] L. Hyafil and R. L. Rivest. Construction optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, 1976.

[39] T. Imielinski and H. Mannila. A database perspective on knowledge discovery. *Communications of ACM*, 39:58–64, 1996.

[40] G. H. John and B. Lent. SIPping from the data firehose. In *Proceedings, Third International Conference on Knowledge Discovery and Data Mining*, pages 199–202. AAAI Press, 1997.

[41] D. Kibler and P. Langley. *Machine Learning as an Experimental Science*, chapter 1, pages 38–43. In [62], 1990.

[42] W. Klösgen and J. Zytkow. Knowledge discovery in database terminology. In U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 573–592. AAAI/MIT Press, 1996.

[43] R. Kohavi. *Wrappers for performance enhancement and oblivious decision graphs.* PhD thesis, Stanford University, September 1995.

[44] J. Kolodner. *Case-based reasoning.* Morgan Kaufmann, San Mateo, CA, 1993.

[45] C.J. Matheus, G. Piatetsky-Shapiro, and D. McNeil. Selecting and reporting what is interesting: The KEFIR application to healthcare data. In U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 495–516. AAAI/MIT Press, 1996.

[46] G. Melli. Ad hoc attribute-value prediction. [1], page 1396.

[47] J. Melton and A. R. Simon. *Understanding the New SQL: A Complete Guide.* Morgan Kaufmann, 1993.

[48] R. S. Michalski, J. G. Carbonell, and T. M. Mitchell. *Machine Learning, An Artificial Intelligence Approach, Vol. 1.* Morgan Kaufmann, 1983.

[49] J. Mingers. An empirical comparison of pruning methods for decision-tree induction. *Machine Learning*, 4:227–243, 1989.

[50] J. Mingers. An empirical comparison of selection measures for decision-tree induction. *Machine Learning*, 3:319–342, 1989.

[51] P. M. Murphy and D. W. Aha. UCI repository of machine learning databases. Irvine, CA: University of California, Department of Information and Computer Science, 1995.

[52] S K. Murthy. *On Growing Better Decision Trees from Data.* PhD thesis, John Hopkins University, 1995.

[53] A. Newell and H. Simon. *Human Problem Solving.* Prentice-Hall, 1972.

[54] G. Piatetsky-Shapiro and W. J. Frawley. *Knowledge Discovery in Databases.* AAAI/MIT Press, 1991.

[55] J. R. Quinlan. Learning efficient classification procedures and their application to chess end-games. In Michalski et al., editor, *Machine Learning: An Artificial Intelligence Approach, Vol. 1*, pages 463–482. Morgan Kaufmann, 1983.

[56] J. R. Quinlan. The effect of noise on concept learning. In Michalski et al., editor, *Machine Learning: An Artificial Intelligence Approach, Vol. 2*, pages 149–166. Morgan Kaufmann, 1986.

[57] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[58] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.

[59] J. R. Quinlan. Combining instance-based and model-based learning. In *Proceedings of the Tenth Internaltional Conference on Machine Learning*, pages 236–243, 1993.

[60] J. R. Quinlan. Improved use of continuous attributes in C4.5. *Journal of Artificial Intelligence Research*, 4:77–90, March 1996.

[61] J. Rissanen. A universal prior for integers and estimation by minimum description length. *Annals of Statistics*, 5(3):416–431, 1983.

[62] J.W. Shavlik and T.G. Dietterich. *Readings in Machine Learning*. Morgan Kaufmann, 1990.

[63] Evangelos Simoudis, Jiawei Han, and Usama Fayyad, editors. *The Second International Conference on Knowledge Discovery and Data Mining*. AAAI Press, August 1996.

[64] P. Smyth and R.M. Goodman. Rule induction using information theory. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, pages 159–176. AAAI/MIT Press, 1991.

[65] P. Smyth and R.M. Goodman. An information theoretic approch to rule induction. *IEEE Trans. Knowledge and Data Engineering*, 4:301–316, 1992.

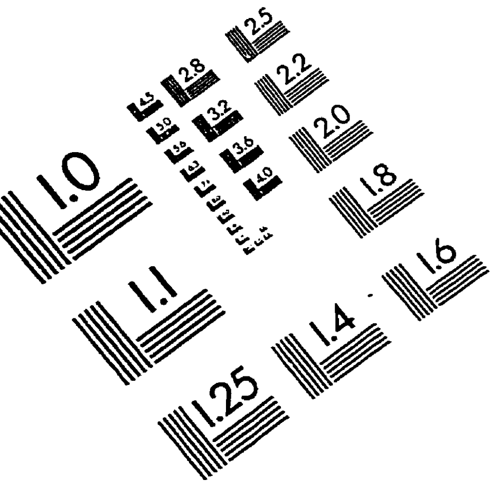
[66] C. Stanfill and D. Waltz. Toward memory-based reasoning. *Communications of the ACM*, 29:1213–1228, 1986.

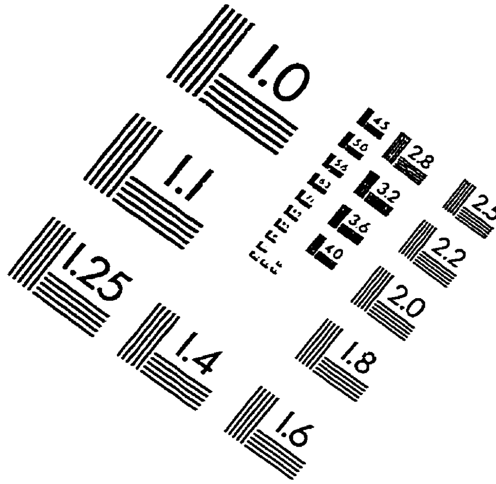
[67] R. Uthurusamy, U. M. Fayyad, and S. Spangler. Learning useful rules from inconclusive data. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, pages 141–157. AAAI/MIT Press.

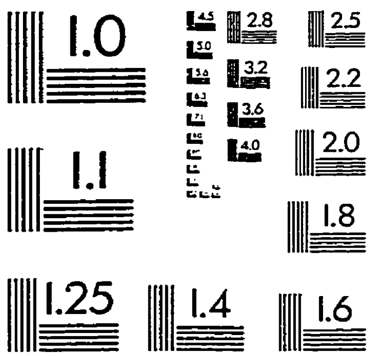
[68] S. M. Weiss and C. A. Kulikowski. *Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems*. Morgan Kaufman, 1991.

[69] P. H. Winston. *Artificial Intelligence*. Addison Wesley, 1992.

IMAGE EVALUATION TEST TARGET (QA-3)

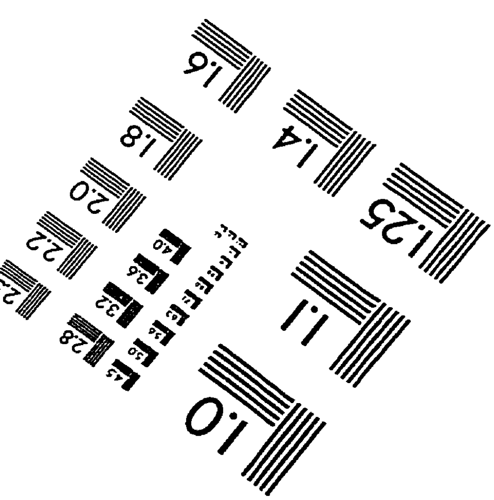


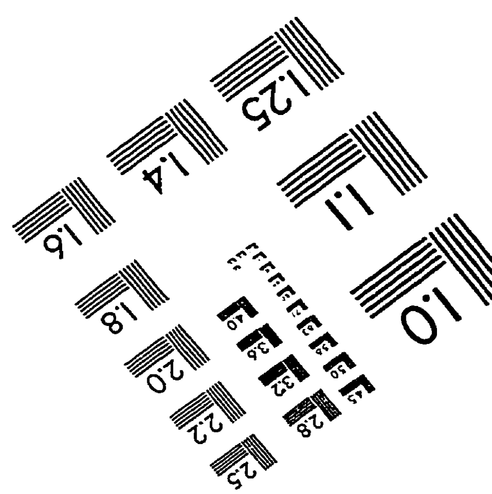




150mm

6"





APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989