SAWTOOTH:
Learning from Huge Amounts of Data

Andrés Sebastián Orrego

Thesis submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Department of Computer Science

Morgantown, West Virginia
2004

Keywords: Incremental Machine Learning, Data Mining, On-line
Discretization.

Abstract

SAWTOOTH: Learning from Huge Amounts of Data

Andrés Sebastián Orrego

Data scarcity has been a problem in data mining up until recent times. Now, in the era of the Internet and the tremendous advances in both, data storage devices and high-speed computing, databases are filling up at rates never imagined before. The machine learning problems of the past have been augmented by an increasingly important one, **scalability**. Extracting useful information from arbitrarily large data collections or data streams is now of special interest within the data mining community. In this research we find that mining from such large datasets may actually be quite simple. We address the scalability issues of previous widely-used batch learning algorithms and discretization techniques used to handle continuous values within the data. Then, we describe an incremental algorithm that addresses the scalability problem of Bayesian classifiers, and propose a Bayesian-compatible on-line discretization technique that handles continuous values, both with a "**simplicity first**" approach and very low memory (RAM) requirements.

To my family.

To Nana.

Acknowledgements

I would like to express my deepest gratitude and appreciation to Dr. Tim Menzies for all of his guidance and support throughout the course of this project. Without his encouragement, his insights, his time and effort this endeavor would not have been possible. His knowledge and motivation have helped me to mature as a student and as a researcher. He has shown great faith in me and my work, which has inspired me to achieve my goal. I admire his dedication, and it has been an honor and a privilege working with him.

In addition, I would like to extend special thanks to Dr. Bojan Cukic. I am grateful for having Dr. Cukic as chair of my Master's Thesis Committee and for his assistance in the pursuit of my project. I would also like to thank Dr. Franz X. Hiergeist, member of my committee and my first advisor at West Virginia University. I appreciate his kindness and assistance throughout my college years.

My sincere gratitude is also expressed to Lisa Montgomery, for her support, encouragement, assistance, guidance and generosity. She has been like a second advisor to me and has made these years at the NASA IV&V Facility a great learning experience.

I would like to thank Dr. Raymond Morehead for helping me to obtain the assistantship which allowed me to pursue this project. Gratefulness is also expressed to West Virginia University, the NASA IV&V Facility, and all my professors throughout these years. My warmest thanks are also given to my friend, Laura, for taking the time to proof-read my work.

Finally, I would like to thank my girlfriend, Nana. Without her love, her patience, and her constant encouragement I would have lost my sanity long before finishing this thesis.

Contents

List of Figures

Chapter 1

Introduction

This thesis addresses the problem of *scalability*, and “incremental learning” in the context of classification and discretization. Scalability refers to the amount of data we can process given an algorithm in a fixed amount of time. It could be thought as the “capacity” of the algorithm in respect to the amount of data it can handle. One approach to scaling up learning is to use incremental algorithms. They have their “answer” ready at every point in time, and update it as new data is processed. We also discuss “concept drift” which relates to the changes in the underlying process or processes that generate the data. Most machine learners assume that data is drawn randomly from a fixed distribution, but in the case of variable processes filling up huge databases, or simulations producing examples (outcomes) over time, this assumption is often violated. For example, the selling performance of a retail store changes drastically depending on promotions, day of the week, weather, etc. We need to be aware of and adapt the learned theory to closely fit the most current state and abandon obsolete concepts in order to achieve the best possible classification accuracy at any given point in time. Even though many algorithms have been offered to handle concept drift, the scalability problem still remains open for such algorithms.

In this thesis we propose a scalable algorithm for data classification from very large data streams based on the widely known, state-of-the-art Naïve Bayes Classifier. The new *incremental* algorithm, called SAWTOOTH, caches small amounts of data (by default 150 examples per cache) and learns from these caches until classification accuracy stabilizes. It is called incremental because it updates the classification model as new instances are sequentially read and processed instead of forming a single model from a collection of examples (dataset) as in batch learning.

After stabilization is achieved, SAWTOOTH changes to *cruise mode* where learning is disabled and the system runs down the rest of the caches, testing the new examples on the theory learned before entering cruise mode. If the test performance statistics ever significantly change, SAWTOOTH enables learning until stabilization is achieved again. SAWTOOTH performance is comparable with the state-of-the-art C4.5 and Naïve Bayes classifier but requiring only one pass over the data even for data sets with continuous (numeric) attributes, therefore, supporting incremental on-line[*] discretiza-

[*]On-line discretization refers to the conversion of continuous values to nominal (discrete) ones while in the process of learning as opposed to a preprocessing step.

tion.

1.1 Motivation

In the real world, large databases are common in a variety of domains. Data scarcity is not the problem anymore, as it used to be years ago. The main issue now is the usage of all available data in the available time, and track possible changes in the underlying distribution of the data to more accurately predict its behavior at each point in time.

In order to make sense of huge or possibly infinite data sets, we need to develop a responsive single-pass data mining system with constant memory footprint. In other words, a linear (or near-linear) time, constant memory algorithm that processes data streams by sequentially reading each instance, updates the learned theory on it or on small sets of them, and then forgets about the processed examples.

The Naïve Bayes classifier offers a good approximation to this ideal. It learns after one read of each instance, it requires very low memory, and it is very efficient for static datasets [?]. One of its drawbacks is that it handles continuous attributes by assuming that they are drawn from a normal distribution, an assumption that may not only be false in certain domains, but, in the context of incremental classification, distributions vary over time.

Our proposed algorithm solves this problem by providing SPADE, an incremental on-line discretization procedure, and a "cruise control" mode that prevents the learned theory to be saturated and over-fitted.

1.2 Goal

The main goal of this research is to develop a **simple** *incremental* classification algorithm that scales to huge or possibly infinite datasets, and is easy to implement, understand, and use. It would serve as a model for scaling up standard data miners using a **simplicity-first** approach.

The algorithm should be able to detect changes in the underlying distribution of data and adapt to those variations. Its performance should be comparable to currently accepted classification algorithms, like the state-of-the-art C4.5 and Naïve Bayes classifier on batch datasets, but with the ability to scale up to unbounded datasets. Overall, it should comply with most, if not all, the following standard data mining goals:

D1 *FAST*: requires small constant time per record, lest the learner falls behind newly arriving data;

D2 *SMALL*: uses a fixed amount of main memory, irrespective of the total number of records it has seen;

D3 *ONE SCAN*: requires one scan of the data and early termination of that one scan (if appropriate) is highly desirable;

D4 *ON-LINE*: on-line suspendable inference which, at anytime, offersthe current "best" answer plus progress information;

D5 *CAN FORGET*: old theories should be discarded/updated if data-generating phenomenon changes;

D6 *CAN RECALL*: the learner should adapt faster whenever it arrives again to a previously visited context.

D7 *COMPETENT:* produces a theory that is (nearly) equivalentto one obtainedwithout the above constraints.

1.3 Contribution

Two main contributions result from this research:

SPADE: A simple incremental on-line discretization algorithm for Bayesian learners that makes possible incremental learning.

SAWTOOTH: A very simple incremental learning algorithm able to process huge datasets with very low memory requirements and performance comparable to standard state-of-the-art techniques.

An extensive literature review is also provided as an aid to understand the field of data mining, particularly in the classification and discretization tasks. Additionally, studies on algorithm evaluation and testing, and findings on data stability are also important contributions to the field.

The proposed algorithms and their implications are explained in this thesis in the following order.

1.4 Organization

The remaining chapters of this thesis are organized as follows.

Chapter two provides a literature review on the topics of classification and discretization. It explains the most relevant algorithms in both fields, giving particular attention and offering a very detailed view of the ones considered the “state-of-the-art”. It investigates these algorithms’ strengths and searches for their possible contributions to incremental learning in distribution changing environments. It further explains why we choose the Naïve Bayes Classifier as the ideal candidate to evolve into an incremental technique. This chapter is of particular importance since it covers the majority of the terms and concepts used later on to describe our creations and findings.

Chapter three gives details on the performance evaluation procedures used for comparing different classifiers. It analyzes different techniques utilized by previous authors and agree on the ones that we believe gives us the statistically best results.

Chapter four unveils SPADE, the first ever one-pass, low memory, on-line, discretization technique for Bayesian classifiers. This new discretizer minimizes information loss and requires one pass through the data. It was developed to handle datasets with continuous attributes so assumptions about the underlying distribution of datasets are avoided. The algorithm is fully described in this chapter and its remarkable results are offered at the end.

Chapter five presents SAWTOOTH, an incremental, low memory Bayesian classifier potentially capable of adapting to changes on the underlying distribution of the data. It integrates the Bayes theory with SPADE and a control on data stability. A study on learning curve stability is also provided in this section. It shows how learning stabilizes early in most cases. This is of particular significance since over-training Bayesian classifiers could result in a saturation of the learned theory. We explain how this impacts the performance of the learner on huge databases and how to solve it in a very simple but powerful way. This study, a section on concept drift, the complete SAWTOOTH algorithm, and a series of case studies are the major components of this chapter.

Conclusions and future work are offered in Chapter six. It summarizes the key issues presented and discusses future work that we think is worth further exploration. Finally, it highlights the major contributions of this thesis to the research area of machine learning.

Chapter 2

Literature Review

2.1 Introduction

Today, in the age of information, data is gathered everywhere. At work, every time a badge is swiped, a log is created and saved into a database. At the supermarket, every transaction is stored electronically. At school, student performance lives in a hard drive. Every day more places are recording our choices and information in an effort to understand us (and the world) better for many different purposes.

Not only is data increasing in size, but also in complexity. As memory becomes cheaper, more and more dimensions of data are recorded in order to get a better snapshot of an event. All this data, millions and millions of terabytes, accumulate in static memory around the world while humans find a way to take advantage of it, understand it, and make it explicit.

2.1.1 Understanding Data

Data mining is the process of discovering patterns that underlie data in order to extract useful information. More formally, it is the extraction of implicit, previously unknown, and potentially useful information from data [?] [?] [?]. Usually, data is stored in databases and data mining becomes the core step for what is called *Knowledge Discovery in Databases* (KDD).

Data analysis requires an effort that is bounded at least to the size of the dataset. The entire dataset has to be read at least once to draw meaningful conclusions from it. For a human, it does not represent a problem when data has few features and a small set of examples, but the analysis quickly becomes impossible as the size and the dimensions of the data increase. Computers can help us solve this problem by *automatically* processing thousands of records per second. Only partial human interaction is necessary to provide a suitable format, and to interpret the computer results.

Machine Learning is the field within artificial intelligence that develops most of the techniques that help us extract information from data by having a machine process the instances, discover patterns in the data, and "learn" a theory to accurately forecast the behavior of new examples. Theories can be of two types, *structural patterns*, or explicit hypothesis, and what we call *tacit hypothesis*. Structural patterns capture the structure of the mined data making it explicit and easy to understand (i.e. decision trees), while tacit theories develop from unclear data processing, but may still accurately make non-trivial predictions on new occurrences (i.e. neural nets). This is why data mining in the field of machine learning is defined as the process of discovering patterns in data, automatically or at least semi-automatically [?].

Many different machine learning approaches and algorithms have been developed since the invention of the computer, some simpler than others. Holte [**?**] offers a case study where a very simple learner performs very well indeed (See §2.2.1). From that case study we adopt its "simplicity first" methodology. Certainly, as we shall see in this thesis, what is true for machine learning in general is just as true for discretization in particular. Our general finding is that very simple discretization works very well as it was suggested by previous authors [**?**]. What is new, is that we can use that "simplicity first" insight to devise a discretization method with properties unavailable in any other method. In particular time and space management for scaling up to infinite datasets. We will center our attention on classification strategies and discretization techniques suitable for this kind of algorithm.

2.1.2 Stream Data

The real world is not the only generator of data. Nowadays, simulations of virtual models are the testing bed for developing real world processes. They are also used to forecast the most likely outcome of an event, providing valuable time to take preventive action. Great amounts of data are generated through this procedure, mainly because they consider a greater number of parameters and outcomes than those produced by actual processes in a very short time. These streams of information, whose storage is impractically expensive, or impossible, have recently been studied in an effort to learn constraints that improve the behavior of the model that generated the data. Numerous experiments with the "incremental treatment learning" approach have shown that setting a small number of variables is often enough to significantly improve the performance of the model [**?**] [**?**] [**?**]. Although applying this technique results in an improved understanding of the model, as explained in [**?**], data mining is unfeasible for large data sets due to the long run-times it requires to perform the overwhelming number of simulation in the Monte Carlo analysis.

In this chapter, we compile the basic definitions and terms offered by authors of previously publicized articles that are closely related to the field of machine learning and our research in particular. First, we introduce relevant Machine Learning approaches, mainly classification learning, and how they differ. Then, we focus on the performance evaluation of classification algorithms and offer a test scenario for comparison among them.

The state-of-the-art classifiers are presented in this review. The remaining algorithms show the evolution of the field of data mining, and serve as a benchmark for newly developed algorithms. The main idea is to offer a representative sample of research in this area.

2.2 Classification

Classifiers comprise one well studied category of machine learners. Their purpose is to determine the outcome of a new example given the behavior of previous examples. As defined by Fayyad, Piatetsky-Shapiro, and Smyth in [?], a classifier provides a mapping function from a data example to one of the possible outcomes. Outcomes are called *classes* and their domain is called the *class attribute*. Examples are called *instances*.

The weather dataset [?] shown in Figure 2.1 consists of fourteen instances where *play* is the class attribute with classes "yes" and "no". Other attributes are *outlook, temperature, humidity, and windy*. The purpose of a classifier is to predict the class value of an unseen instance, that is, whether or not we play golf given the weather of a new day. This dataset will be used as an example throughout this chapter.

Instance	*Attributes*				*Class*
	outlook	temperature	humidity	windy	play
1	sunny	hot	high	false	no
2	sunny	hot	high	true	no
3	overcast	hot	high	false	yes
4	rainy	mild	high	false	yes
5	rainy	cool	normal	false	yes
6	rainy	cool	normal	true	no
7	overcast	cool	normal	true	yes
8	sunny	mild	high	false	no
9	sunny	cool	normal	false	yes
10	rainy	mild	normal	false	yes
11	sunny	mild	normal	true	yes
12	overcast	mild	high	true	yes
13	overcast	hot	normal	false	yes
14	rainy	mild	high	true	no

Figure 2.1: The weather dataset with all discrete attributes.

Accurate forecasting is possible by analyzing the behavior of previous observations or instances and building a model for prediction. Measuring the performance of classification involves two main phases, the *training phase* and the *testing phase.* During training, the classifier processes the known examples called *training instances*, and builds the model based on their class information. This is known as *supervised learning.* However, in *unsupervised learning*, the class labels are either absent or not taken into consideration for the training process. After training, the classifier tests the learned model against the *test dataset.* Instances from the test dataset are assigned a class according to the model and each prediction is then compared to the actual instance class. Each correctly classified instance is counted as a *success* and the remaining are *errors.* The accuracy or *success rate* is the proportion of successes over the whole set of instances. Similarly, the *error rate* is the ratio of errors over the total number of examples. If an acceptable accuracy is achieved, then the learned model is used to classify new examples where the class is unknown.

Classification models may be described in various forms: *classification rules, decision trees, instance based learning, statistical formulae, or neural networks.* We will focus special attention to decision tree induction and Statistical formulae.

2.2.1 Decision Rules

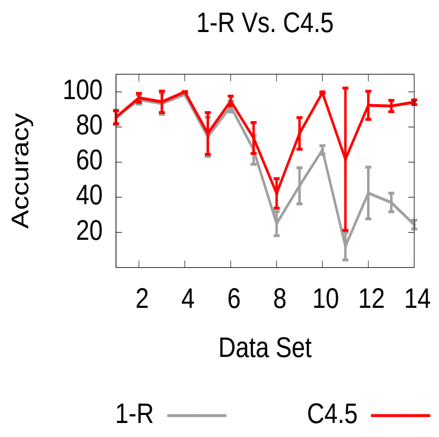
We start with this topic because rudimentary decision rules can be extracted easily using very simple algorithms. Also, *simplicity* is one of the “slogans” of this thesis and should always be tried first.

One of the simplest learners ever developed is *“1-R”* which stands for *one-rule*. Holte presented *1-R* in the paper, *“Very simple classification rules perform well on most commonly used datasets”* [?]. 1-R reads a dataset and generates a one-level decision tree described by a set of rules always testing the same attribute. Its set of rules is of the form:

```
IF $Attribute_i = Value_1$   THEN $Class = Max(class_{i1})$
ELSE
IF $Attribute_i = Value_2$   THEN $Class = Max(class_{i2})$
ELSE
⋮
ELSE
IF $Attribute_i = Value_n$   THEN $Class = Max(class_{in})$
```

$$Where \begin{cases} Attribute_i & = \text{Selected Attribute.} \\ Value_{1\ldots n} & = \text{One of the } n \text{ different values of } Attribute_i. \\ Max(class_{i,n}) & = \text{The majority class for the attribute-value } i, n. \end{cases}$$

Surprisingly, 1-R sometimes achieves very high accuracies suggesting that the structure of many real-world datasets are simple and many times it depends on just one highly influential attribute. An accuracy comparison between 1-R and the state-of-the-art decision tree learner C4.5 (§2.2.2) on fourteen datasets from the University of California Irvine (UCI) repository [?] is presented in Figure 2.2.



Number	Dataset Name
1	Credit-a
2	Vote
3	Iris
4	Mushroom
5	Lymph
6	Breast-w
7	Breast-cancer
8	Primary-tumor
9	Audiology
10	Kr-vs-Kp
11	Letter
12	zoo
13	Soybean
14	Splice

Figure 2.2: Accuracy comparison between 1-R and the more complex decision tree learner C4.5

These are the results from a tenfold cross-validation (see §3.2.1) on each of the datasets and sorted by the accuracy difference between 1-R and C4.5. In more than 50% of the datasets, 1-R's performance is very close to C4.5's, but in some of the remaining datasets there is enough "big" difference in the results to encourage us to look further than 1-R.

The algorithm for 1-R divides up into three main parts. First, it generates a different set of rules for each attribute, one rule per attribute value. Then, it tests each attribute's rule set and calculates the error rate. Finally, it selects the attribute with the lowest overall error rate – in the case of a tie, it breaks it arbitrarily – and proposes its rules as the theory learned. The pseudo-code for 1-R is depicted in Figure 2.3.

```
1.  for each attribute $A$ {
2.     for each value $V_A$ {
3.     Count class of $V_A$ occurrences
4.        $V_A \leftarrow$Max(Class)
5.     }
6.     $Error_A \leftarrow$ Test $V_A$ against the whole dataset
7.  }
8.  Select attribute with Min($Error_A$)
```

Figure 2.3: 1-R pseudo-code

Simple and modestly accurate, it also handles missing values and continuous attributes. Missing values are dealt with as if they were another attribute's value; therefore, generating an additional branch (rule) for the value *missing*. Continuous attributes are transformed into discrete ones by a procedure explained in the discretization section §2.3 of this chapter.

As stated before, *1-R* encourages a *simplicity first* methodology, and serves as a baseline for performance and theory complexity of more sophisticated classification algorithms.

2.2.2 Decision Tree Learning

After his invention of ID3 [?] and the *state-of-the-art* C4.5 [?] machine learning algorithms, Ross Quinlan became one of the most significant contributors to the development of classification. Both classifiers model theories in a tree structure.

Following a *greedy*, *divide-and-conquer*, *top-down* approach, decision tree learners choose one attribute to place at the root node of the tree and generate a branch for each attribute value, effectively splitting the dataset into one subset per branch. This process is repeated recursively for each subset until all instances of a node belong to a single class or until no further split is possible. Selecting a criterion to pick the best splitting attribute is the main decision to be made. The smaller trees can be built by selecting the attribute whose splits produce leaves containing instances belonging to only one class, that is, the purest daughter nodes [?]. The *purity* of an attribute is called *information* and is measured in *bits*.

Entropy is the information measure usually calculated in practice. As an example, let D be a set of d training examples, and $C^i(i = 1 \cdots n)$ be one of n different classes. The information needed to classify an instance or the entropy of D is calculated by the formula:

$$E(D) = -\sum_{k=1}^{n} p_i \log_2(p_i) \qquad \text{where} \qquad p_i = \frac{|D, C^i|}{|D|} \tag{2.1}$$

p_i is the probability of the class C_i in D. This formula can be translated to the more used:

$$E(D) = \frac{(-\sum_{i=1}^{n} |C_i| \log_2 |C_i|) \quad + \quad |D| \log_2 |D|}{|D|} \tag{2.2}$$

$|C_i|$ is the number of occurrences of Class C_i in the dataset D. $|D|$ is the size of the dataset D. This is the information of the dataset as it is, without splitting it. Now let us suppose an attribute A with m different values is chosen to split the data set D. D is partitioned producing subsets $D_j(j = 1 \cdots m)$. The entropy of such split is given by:

$$E(A) = \sum_{j=1}^{m} \frac{|D_j|}{|D|} * E(D_j) \tag{2.3}$$

Where $E(D_j)$ is calculated as described in Equation 2.2. The entropy of a pure node D_j is 0 *bits*.

Now, the *information gained* by splitting D in attribute A is the difference between the entropy of the data set D (Equation 2.2) and the entropy of the split in the attribute A (Equation 2.3), that is:

$$InfoGain(A) = E(D) - E(A).$$

This *information gain* measure is evaluated for each one of the attributes left for selection. Maximizing $InfoGain$ is ID3's main criterion for choosing the splitting attributes at every step of the tree construction.

One problem of this approach is that the information gain measure favors attributes having large possible values, generating trees with multiple branches and many daughter nodes. This is best seen in the extreme case that an attribute contains a single different value per instance. Let us say that in Figure 2.1 *instance* (the example number) is an attribute. There is one different *instance* value per entry. If we calculate the entropy of such attribute according to Equation 2.3, we have $E(instance) = 0$ bits, since such a split would branch into pure nodes. The entropy of the dataset would be:

$$E(weather) = E([9,5]) = [-9\log_2 9 - 5\log_2 5 + 14\log_2 14]/14 = 0.940\text{bits}$$

then, the attribute‘s *information gain* would be: $InfoGain(instance) = E(weather) - E(instance) = 0.0940$ bits, which is higher than the ones from all the other attributes. Therefore, ID3 would generate a 1-level deep tree placing the attribute *instance* at the root. Such a tree tells us nothing about the structure of the data and does not allow us to classify new instances, which are the two main goals of classification learning.

C4.5 partially overcomes this bias by using a more robust measure called *gain ratio*. It simply adjusts the information measure by taking into account the number and size of the daughter nodes generated after branching on an attribute. The formula for gain ratio is:

$$GainRatio(A) = \frac{InfoGain(A)}{E([|D_1|, |D_2|, \ldots, |D_m|])}$$

Unfortunately, it is reported that the gain ratio fix carries too far and can lead to bias toward attributes with lower information than the others [**?**] [?].

Going back to the weather dataset, its decision tree structure is depicted in Figure 2.4. Nodes are attributes and leaves are the classes. Each branch is a value of the attribute at the parent node. Predictions under this structure are made by testing the attribute at the root of the tree, and then recursively moving down the tree branch corresponding to the value of the attribute until a leaf is reached. As an example, let's suppose we have a new instance *outlook = sunny, temperature = cool, humidity = high, windy = true.* If we want to classify this instance, we start at the root testing the outlook attribute. Since outlook = sunny in the new instance, then we follow the left branch and arrive at humidity. We test humidity and we find that it is "high", therefore we keep following the left branch and arrive at a leaf the class "no", suggesting that we do not play golf under the above circumstances.

Following the same classification procedure we find that this tree has 100% accuracy on the training data, in other words, all 14 training instances are classified correctly under the described model. It is not surprising to achieve very high accuracies when testing on the training data. It is analogous to predicting yesterday's weather today. Measuring a learner's performance on old data is not a good predictor of its performance on future data.

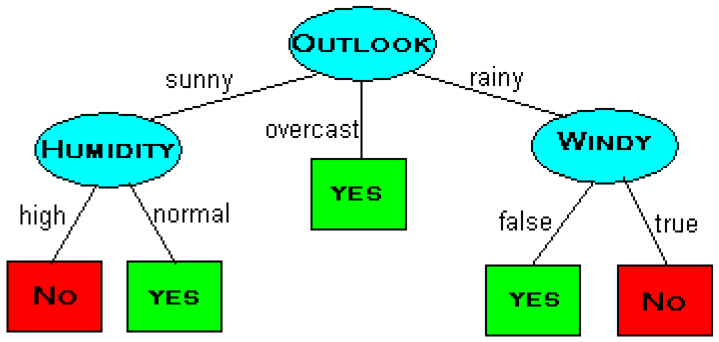


Figure 2.4: Decision Tree for the weather data.

Classification targets new instances, instances where the class is unknown; therefore a better approximation to the true learner's performance is to asses its accuracy on a dataset completely different than the training one. This independent dataset is known as the *test data*. Both training and test sets need to be representative in order to give the true performance on future data. Normally, the bigger the training set, the better the classifier. Similarly, the bigger the test set, the better the approximation to the true accuracy estimate. When plenty of data is available, there is no problem in separating a representative training set and test set, but when data is scarce, the problem becomes how to make the most of the limited dataset.

2.2.3 Other Classification Methods

Covering Decision Rules

More classification rule learners have been developed and their theories are usually described using a bigger, more complex set of rules. *Covering Algorithms*, like the one called *Prism*, usually test on more than a single attribute and sometimes their rules contain conjunctions and disjunctions of many preconditions in an effort to cover all the instances belonging to a class and, at the same time, excluding all instances that do not belong to it. Building a theory from such algorithms is simple: First, Prism creates the rule with the empty left-hand-side and one of the classes as the right-hand-side. For example:

if ? **then** $Class = c$.

This rule covers all the instances in the dataset that belong to the class. Then, Prism restricts the rule by adding a test as the first term of the left-hand-side (L.H.S.) of the equality. It keeps adding attribute values to the L.H.S. until the accuracy of the rule is 100%, therefore creating only "perfect" rules.

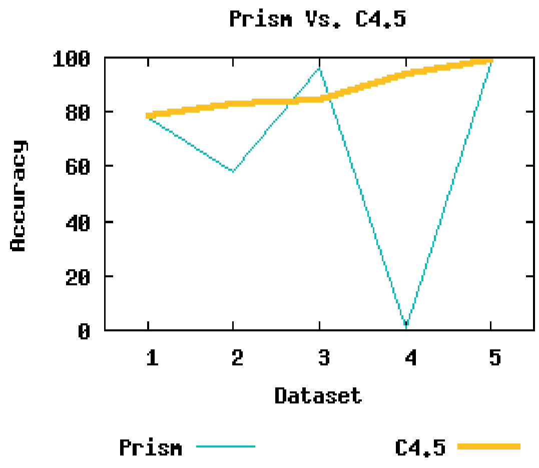
Prism's classification performance on new data is very limited since it only relies on the classification of the training data to create its theory. Prism's rules fit very well the known instances, but not very well the unknown ones (testing set). This phenomenon is known as *overfitting* and can be caused by this and many other reasons as we shall see in this thesis.

A comparison between Prism and C4.5 on classification accuracy is depicted in Figure 2.5. This graph shows the results of a tenfold cross-validation procedure on each dataset and for both classifiers. These results are better estimates of the accuracy on new data as we shall see in §3.2.1.

Decision rules are an accepted but inferior alternative to the more complex theory learned by decision tree induction. However, the algorithmic complexity of decision rule learners (several passes through the training data) makes them unsuitable for incremental learning. Decision trees are explained next.

Instance Based Classification

The idea behind *Instance-based learning* is that under the same conditions an event tends to have the same consequences at every repetition. Further,



Number	Dataset Name
1	Lymph
2	Contact-lenses
3	Tic-tac-toe
4	Splice
5	Kr-vs-kp

Figure 2.5: Accuracy comparison between Prism and the state-of-the-art C4.5. Prism's accuracy instability is due to overfitting.

the consequence (class) of an event (instance) is closest to the outcome of the most similar occurrence(s). An instance-based learner just has to memorize the training instances and then go back to them when new instances are classified. This scheme falls into a category of algorithms known as *lazy learners*. They are called *lazy* because they delay the learning process until classification time.

Training the learner consists of storing the training examples as they appear, then, at classification time, a distance function is used to determine which instance of the training set is "closest" to the new instance to be classified. The only issue is defining the distance function.

Most instance-based classifiers employ the Euclidian distance as the distance function. The Euclidian distance between two points $X = (x_1, x_2, \ldots, x_n)$

and $Y = (y_1, y_2, \ldots, y_n)$ is given by the formula:

$$\mathrm{d}(X, Y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

With this formula, the class value of the training instance that minimizes the distance to the new example becomes the class of the new example. This approach is known as the *Nearest Neighbor* and has the problem of being easily corrupted by noisy data. In the event of outliers in the instance space, new instances closest to these points would be misclassified. A simple but time consuming solution to this problem is to cast a vote from a small number k of nearest neighbors and classify according to the majority vote. Another problem is that distance can be measured easily in a continuous space, but there is no immediate notion of distance in a discrete one. Nearest neighbor methods handle discrete attributes by assuming a distance of 1 for values that differ, and a distance of 0 for values that are the same.

Nearest Neighbor instance-based methods are simple and very successful classifiers. They were first studied by statisticians in the early 1950's and then introduced as classification schemes in the early 1960's. Since then they remain among the most popular and successful classification methods, but their high computational cost when the training data is large prevents them from developing into more sophisticated incremental learning algorithms.

Neural Networks

Inspired by the biological learning process of nervous systems, artificial neural networks (ANN) [?] simulate the way neurons interact to acquire knowledge. Neural nets learn similarly to the brain in two respects [?]:

- Knowledge is acquired by the network through a learning process.
- Interconnection strengths known as synaptic weights are used to store the knowledge.

The network is composed of highly interconnected processing elements, called *neurons*, working parallel to solve a specific problem. Each connection between neurons has an associated weight that is adjusted during the training phase. After this phase is finished and all the weights are adjusted, the architecture of the neural network becomes static and ready for the test phase. In this later phase, instances visit the neurons according to the net's architecture, they are multiplied by the neuron's associated weights, some calculations take place, and an outcome is produced. This output value is then passed to the next neuron as input and the process is repeated until an output value determines the instance class.

Depending on the connections between neurons, various ANN models have been developed. They can be sparsely-connected, or fully-connected as in the Hopfield Networks [?]. They could also be *recurrent* such as Boltzmann Machines [?] in which the output of a neuron not only serves as the input to another but also feedbacks itself. Multilayered feed-forward networks, like

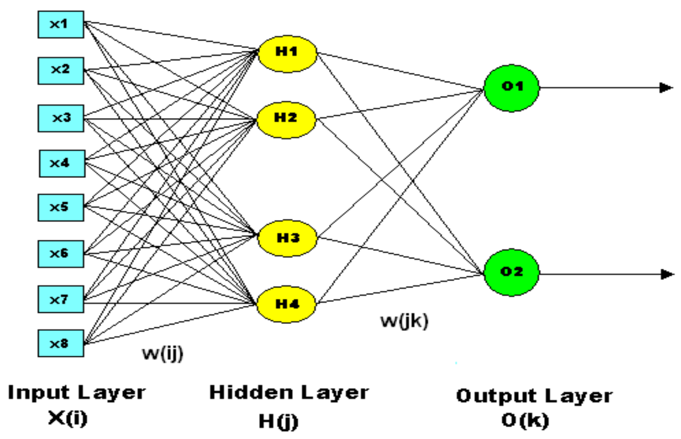


Figure 2.6: Fully-connected feed-forward neural network with one hidden layer and one output layer. Connection weights between nodes in the input and the hidden layer are denoted by $w(ij)$. Connection weights between neurons in the hidden and the output layer are $w(jk)$

the *multilayered percepton* shown in Figure 2.6, are the most widely used. They have an input layer of source nodes, an output layer of neurons, and layer of hidden neurons that are inaccessible to the outside world. The output of one set of neurons serves as the input for another layer.

ANN training can be achieved by different techniques. One of the most popular is known as *Backpropagation*. It involves two stages [**?**]:

- *Forward stage.* During this stage the free parameters of the network are fixed and instances are iteratively processed layer by layer. The difference between the output generated and the actual class is calculated and stored as the error signal.

- *Backward stage.* In this second stage the error signal is propagated backwards from the output layer down to the first hidden layer. During this phase, adjustments are applied to the free parameters of the network, using gradient descent, to statistically minimize the error.

A major limitation of back-propagation is that it does not always converge [?] or convergence might be very slow, therefore, training could take a very long time or it may perhaps never end.

ANN main advantages are the tolerance to noisy data and their particular ability to classify a pattern on which they have not been trained. On the other hand, one major disadvantage is their poor interpretability: additional utilities are necessary to extract a comprehensible concept description.

Although they can perform better than other classifiers [?], ANN long time requirements make them unsuitable for learning on huge data sets. Also, their complex theories go against our goal of simplicity, therefore, we do not discuss them any further.

Treatment Learning

Neural networks learning algorithms and other miners look for complex and detailed descriptions of concepts learned to better fit the data and more accurately predict future outcomes. However, such learning is unnecessary in domains that lack complex relationships [?] [?]. Treatment learning was developed with *simplicity* in mind.

Created by Menzies and Hu [?], treatment learning (or *Rx Learning*) mines minimal contrast set with weighted classes [**?**]. It does not classify, but finds conjunctions of attribute-value pairs, called *treatments*, that occur more frequently under the presence of preferred classes and less frequently under the presence of other classes. It looks for the treatment that better selects the best class while filtering out undesired classes.

The concept of best class derives from the assumption that there is a partial ordering between classes, that is, each class has some weight or score that determines its priority over the rest. The *best class* is the one considered superior than the other ones and, therefore, has the highest weight. Similarly, the *worst class* is the least desirable and has the lowest score. The class values are determined by the user or by a scoring function depending on the domain and the goal of the study.

Tar2 is the first known treatment learner made available to the public. This software along with documentation can be downloaded from `http://menzies.us/rx.html`. It produces treatments of the form:

$$\boxed{\begin{array}{l}\textbf{if } Rx : Att_A = Val_{Az} \wedge Att_B = Val_{By} \ldots \\ \textbf{then } class(C_i) : confidence(Rxw.r.t.C_i)\end{array}}$$

Where *confidence* of a treatment with respect to a particular class C_i is the conditional probability of C_i on the instances selected by the treatment. That is:

$$confidence(Rxw.r.t.C_i) = P(C_i|Rx) = \frac{|examples \in (Rx \wedge C_i)|}{|examples \in Rx|} \tag{2.4}$$

Good treatments have significantly higher confidence in the best class and significantly lower confidence in the worst class than the original distribution.

The best treatment is the one with the highest lift, that is, the one that most improves the outcome distributions compared to the baseline distribution. In the case of the weather example, $outlook = overcast$ is the best treatment since it always appears when we play golf and never emerges when we play no golf.

The impact of Tar2's output on the class distribution is depicted in Figure 2.7.

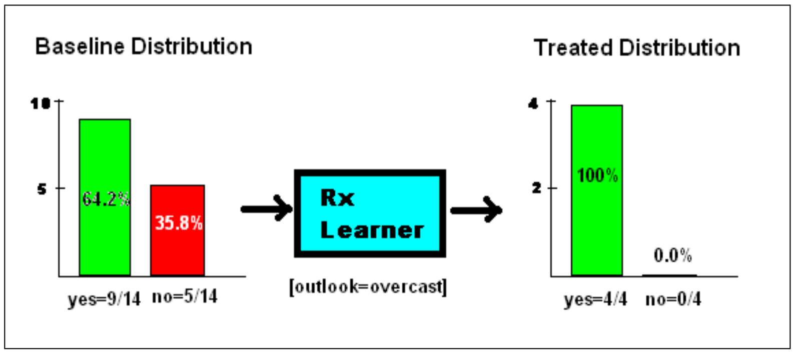


Figure 2.7: Class frequency of the dataset before and after being treated.

Treatment learning offers an attractive solution for monitoring and controlling processes. Several studies have demonstrated its usefulness in parameter-tuning and feature subset selection [?]. Even though the last version of the *TAR*s, Tar3, has improved the learning time by adding heuristic search, it is still polynomially bounded in time and requires the storage of training instances in memory.

2.2.4 Naïve Bayes Classifier

The Naïve Bayes Classifier (NBC), is a well studied probabilistic induction algorithm that evolves from work in pattern recognition [?]. This statistical supervised approach allows all the attributes to contribute equally to the classification and assumes that they are independent of one another. It is based on the Bayes theorem which states that the probability of an event E resulting in a consequence C can be calculated by dividing the probability of the event given the consequence times the probability of the consequence by the probability of the event. In other terms:

$$P[C|E] = \frac{P[E|C] \times P[C]}{P[E]}. \tag{2.5}$$

In the context of classification, we want to determine the class value given an instance. The Bayes theorem could help us determine the probability that an instance i, described by attribute values $A_1{=}V_{1,i} \wedge \ldots \wedge A_n{=}V_{n,i}$, belongs to a class C_j. Going back to Equation 2.5:

$$P(C_j|A_1{=}V_{1,i} \wedge \ldots \wedge A_n{=}V_{n,i}) = \frac{P(C_j) \times P(A_1{=}V_{1,i} \wedge \ldots \wedge A_n{=}V_{n,i}|C_j)}{P(A_1{=}V_{1,i} \wedge \ldots \wedge A_n{=}V_{n,i})}. \tag{2.6}$$

where $P(C_j|A_1{=}V_{1,i} \wedge \ldots \wedge A_n{=}V_{n,i})$ is the conditional probability of the class C_j given the instance i; $P(C_j)$ is the number of occurrences of the class C_j over the total number of instances in the dataset, also known as the *prior* *probability* of the class C_j; $P(A_1{=}V_{1,i} \wedge \ldots \wedge A_n{=}V_{n,i}|C_j)$ is the conditional probability of the instance i given the class C_j; and $P(A_1{=}V_{1,i} \wedge \ldots \wedge A_n{=}V_{n,i})$ is the prior probability of the instance i. In order to minimize the classification error the most likely class is chosen for classification, that is, $argmax_{C_j}(P(C_j|A_1{=}V_{1,i} \wedge \ldots \wedge A_n{=}V_{n,i})$ for each instance i [?] [?] [?] [?]. Since the denominator in Equation 2.6 is the same across all classes, it can be omitted as it does not affect the relative ordering of the classes, therefore:

$$P(C_j|A_1{=}V_{1,i} \wedge \ldots \wedge A_n{=}V_{n,i}) = P(C_j) \times P(A_1{=}V_{1,i} \wedge \ldots \wedge A_n{=}V_{n,i}). \quad (2.7)$$

Since i is usually an unseen instance, it may not be possible to directly estimate $P(A_1{=}V_{1,i} \wedge \ldots \wedge A_n{=}V_{n,i}|C_j)$. Based on the assumption of attribute conditional independence, this estimation could be calculated by:

$$P(A_1{=}V_{1,i} \wedge \ldots \wedge A_n{=}V_{n,i}|C_j) = \prod_{k=1}^{n} P(A_k = V_{k,j}|C_j). \quad (2.8)$$

Finally, combining Equation 2.7 and Equation 2.8 results in:

$$P(C_j|A_1{=}V_{1,i} \wedge \ldots \wedge A_n{=}V_{n,i}) = P(C_j) \times \prod_{k=1}^{n} P(A_k = V_{k,j}|C_j). \quad (2.9)$$

NBC uses Equation 2.9. This equation can be solved by maintaining a very simple counting table. During training, every attribute value occurrence is recorded along with its class by incrementing a counter, the number of counters depends on the number of attribute values and the number of classes. Probabilities can be then calculated from the table and classification can be easily performed. As an example, let us go back to the weather dataset. Figure 2.8 presents the summary of the weather data in terms of counts.

outlook			temperature			humidity			windy			play	
	yes	no		yes	no		yes	no		yes	no	yes	no
sunny	2	3	hot	2	2	high	3	4	false	6	2	9	5
overcast	4	0	mild	4	2	normal	6	1	true	3	3		
rainy	3	2	cool	3	1								

Figure 2.8: Bayesian Table for the weather dataset

From the Figure 2.8 we can see that for each one of the attribute value pairs there are two counters, one for *play* = *yes* and one for *play* = *no*. These counters are incremented every time an attribute value of an instance is seen. For instance, *temperature* = *mild* occurred six times, four times for *play* = *yes* and two times for *play* = *no*. We also note that overall, nine times *play* = *yes* and five times *play* = *no*, as summarized in the rightmost columns. From this information we can build the probabilities necessary to classify a new day. For example, classifying the new instance depicted in Figure 2.9 involves the following:

outlook	temperature	humidity	windy	play
rainy	hot	high	true	?

Figure 2.9: A New Day

$$P(yes|rainy \wedge hot \wedge high \wedge true) = P(rainy|yes)P(hot|yes)P(high|yes)P(true|yes)P(yes)$$

$$= (3/9) \times (2/9) \times (3/9) \times (3/9) \times (9/14) = 0.005291 \quad (2.10)$$

$$\begin{aligned} P(no|rainy \land hot \land high \land true) &= P(rainy|no)P(hot|no)P(high|no)P(true|no)P(no) \\ &= (2/5) \times (2/5) \times (4/5) \times (3/5) \times (5/14) = 0.027429 \quad (2.11) \end{aligned}$$

From the results in Equation 2.10 and Equation 2.11 we can conclude that Naïve Bayes would classify the new day as $play = no$.

As we see from the previous example, calculating probabilities from discrete feature spaces is straightforward. Counting occurrences of few values per attribute is very efficient. Continuous attributes often have finite but large number of values, where each value appears in very few instances, thus generating a large number of small counters. Sampling probabilities from such small spaces gives us unreliable probability estimation, leading to poor classification accuracy. NBC handles continuous attributes by assuming they follow a Gaussian or *normal* probability distribution. The probability of a continuous value is estimated using the probability density function for a normal distribution which is given by the expression

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (2.12)$$

Where μ is the mean value of the attribute space, and σ is the standard deviation. The mean and standard deviation for each class and continuous attribute is calculated from the *sum* and sum^2 of the attribute, given the class. This means that NBC can still incrementally learn, even in the presence of continuous attributes.

It is important to note that the probability density function of a value x, $f(x)$, is not the same as its probability. The probability of a continuous attribute being a particular continuous value x is zero, but the probability that it lies within a small region, say $x \pm \epsilon/2$, is $\epsilon \times f(x)$. Since ϵ is a constant that weighs across all classes, it will cancel out, therefore, the probability density function can be used in this context for probability calculations.

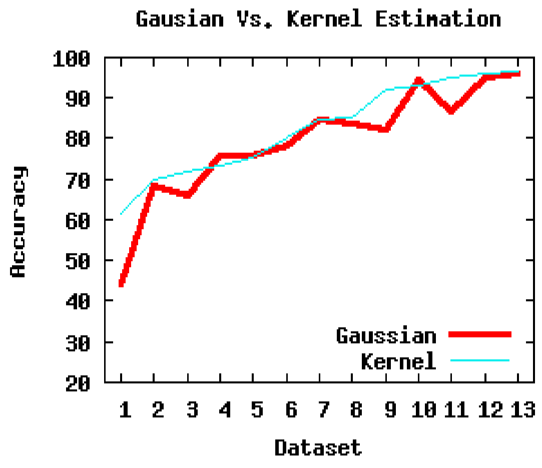
When classifying a new instance, the probability of discrete values is calculated directly from the counters within the Bayesian table, while the probability of a continuous value x is estimated by plugging the values x, μ, and σ into the probability density function. The result is then multiplied according to Bayes' rule.

Only one pass through the data and simple operations are necessary to keep the table updated, making NBC a very simple, efficient, and low-memory requirement algorithm suitable for incremental learning. Although it carries some limitations, mainly induced by the attribute independence assumption often violated in real-world datasets [?], it has been shown that it is also effective and robust to noisy data [?] [?] [?] [?] [?].

Kernel Estimation

The probability estimation of an event is nothing more that that, an estimation. The probability density function of the normal distribution provides a reasonable approximation to the distribution of many real-world datasets, but it is not always the best [?]. In datasets that do not follow a normal distribution, the classification accuracy could be improved by providing more general methods for density estimation. This problem has been the subject of numerous studies aiming for better probability estimation so the classification accuracy becomes optimal.

John and Langley in [?] investigate *kernel estimation* with Gaussian kernels, where the estimated density is averaged over a large set of kernels. For this procedure to take place, all the n values of a continuous attribute must be stored. Estimating the probability density function of a value for the classification of a new instance requires the evaluation of the Gaussian probability density function (Equation 2.12) n times - once per continuous value – with $\mu =$ to the current value and $\sigma = \frac{1}{\sqrt{n_c}}$ where n_c is the number of instances belonging to class c. The average of the evaluations becomes the probability estimation for a value. This process could be thought as piling up n small Gaussian probabilities with varying height according to each one of the attribute values and with constant width (σ). This practice discovers distribution skews and approximates better to the actual distribution. In practice, Naïve Bayes Classifier with kernel estimation, denoted *NBK* in this thesis, performs at least as well as Naïve Bayes with Gaussian assumption (NBC), and in some domains it outperforms the Gaussian assumption. Figure 2.10 shows the performance of the Naïve Bayes classifier under both assumptions.



Number	Dataset Name
1	Vehicle
2	Horse-colic
3	Vowel
4	Auto-mpg
5	Diabetes
6	Echocardiogram
7	Heart-c
8	Hepatitis
9	Ionosphere
10	Labor
11	Anneal
12	Hypothyroid
13	Iris

Figure 2.10: Accuracy comparison between Gaussian estimation (NBC) and Kernel Estimation (NBK) for the Naïve Bayes classifier

The drawback of kernel estimation is its computation complexity. Since it requires continuous attribute values to be stored the NBK becomes unsuitable for incremental learning and is no longer memory efficient.

Another approach for handling continuous attributes that works for discrete space learners including the Naïve Bayes Classifier is called *discretization*. Although discretization techniques convert continuous feature spaces to discrete ones independently of the learning scheme, our focus is discretization performance under the Naïve Bayes Classifier, mainly because it is a learner apt for incremental classification.

2.3 Discretization

Discretization techniques are developed and widely studied not only because many machine learning algorithms require a discrete space, but because they may improve the accuracy and performance achieved by some others that support continuous features [?] [**?**]. Many discretization methods have been developed through the years, many of them focusing on minimizing the error of the learned hypothesis. In this literature review we will refer to the state-of-the-art discretization methods and the research that is behind them.

2.3.1 Data Preparation

Machine learning from raw data in any format and any size is not possible so far. Researchers and Data Miners have to follow some steps prior to learning [**?**]. Formatting the data is one tedious, time consuming, and unavoidable preprocessing step [?]. In this data transformation process, data is shaped into a specific format so the learner can interpret it correctly. Machine learner implementations require some kind of differentiation between fields and records (instances). They also need to be able to match each field of an instance with an attribute of the dataset. One way to address this problem is to store data in a table like file where each line is an instance and fields are bounded by a field separator which could be a comma, a tab, a space, etc. Another way could be XML format. In any case, a predefined way of reading the data is necessary and converting to it is a must. Some learners even require the attribute values in the data to be known before hand. Those usually expect a header on the data file or a header file containing information about the data such as the name of the dataset, the types of the attributes and/or the attribute values present in the data. Another problem with raw data is the presence of unsuitable or sometimes mixed attribute types.

Attribute Types

According to many authors, there are several kinds of attribute domains. In [?] John and Langley classify attributes into either *discrete* or *numeric*, while Yang and Web in [?] talk about *categorical* versus *numeric* where numeric attributes can be either continuous or discrete. Witten and Frank in [?] offer a better differentiation between attribute types that we summarize here for consistency throughout this thesis. We will classify attribute types in two main groups, *qualitative* and *quantitative*. Qualitative attributes refer to characteristics of the data and their values are distinct symbols forming labels or names. Two subcategories can be derived from it. *Nominal* values have no ordering or distance measure. Examples of nominal attributes are:

- Outlook: sunny, overcast, rainy.
- Blood Type: A, B, O, AB.

Ordinal values have a meaningful logical order and can be ranked, but no arithmetic operations can be applied to them. Examples of ordinal attributes are:

- Temperature: $cool < mild < hot$.
- Student evaluation: $excellent > good > pass > fail$.

Quantitative attributes measure numbers and are numeric in nature. They possess a natural logic order therefore can be ranked. Also, meaningful arithmetic operations can be applied to their values, and their distance can be measured. Two subcategories of quantitative attributes can be distinguished based on the *level of measurement.* In the *interval* level of measurement data can be ranked, and distance can be calculated. Some, but not all arithmetic operations can be applied to this type of attribute mainly because *zero* in the interval level of measurement does not mean 'nothing' as *zero* in arithmetic. A good example of interval level measurement attribute is dates in years. Having various values (years) such as $V_1 = 1940$, $V_2 = 1949$, $V_3 = 1980$, and $V_4 = 1988$, we can order them ($V_1 < V_2 < V_3 < V_4$), we can calculate the difference between the years V_1 and V_4 ($1988 - 1940 = 48$ years), and we can even calculate the average of the four dates (1964), and all those calculations make sense. What would not make much sense is to have five times the year V_1 ($1940 \times 5 = 9700$) or the sum between the years V_2 and V_3 (62864) because the year 0 is not the first year we can start counting from, it is an arbitrary starting point.

The last level of measuring is the *ratio* quantities. This level inherently defines a starting point *zero* that, the same as the arithmetic *zero*, means "nil", or "nothing". Also, any arithmetic operation is allowed. An example of ratio attributes could be "the number of male students in a classroom." A classroom can have three times the number of male students of another classroom, or *zero* (or no) male students.

Ratio is the most powerful of the measuring levels in terms of information. Interval, ordinal, and nominal follow in that order. Conversion from a higher level to a lower one will lose information (generalization). Figure 2.11 gives a summary of the characteristics of the different attribute types from the most informative down to the least.

TYPE	LEVEL	LOGICAL ORDER	ARITHMETIC	ZERO
Quantitative	Ratio	yes	any	defined
Quantitative	Interval	yes	some	not defined
Qualitative	Ordinal	yes	none	not defined
Qualitative	Nominal	no	none	not defined

Figure 2.11: Level of measurement for attributes

For simplicity, we will only differentiate between the two broader categories of attribute types: *qualitative* and *quantitative*. From now and for the remainder of this thesis, we will call qualitative attributes *discrete*, and quantitative attributes will be denoted *continuous*.

2.3.2 Data Conversion

Handling continuous attributes is an issue for many learning algorithms. Many learners focus on learning in discrete spaces only [?] [**?**]. The presence of a vast number of different values for an attribute and few occurrences for each of its values will increase the likelihood that instances will have the same class as the majority class creating what is known as *overfitting*. Overfitting guarantees high accuracy when testing on the training dataset but poor performance on new data, therefore treating continuous values as discrete is discouraged. Discretization is the process by which continuous attribute values are converted to discrete ones, often grouping consecutive values into one. Several discretization techniques have been developed throughout the years for different purposes. Discretization is not only useful for those algorithms that require discrete features, but it also facilitates the understanding of the theories learned by making them more compact, increases the speed of induction algorithms [?], and increases the classification performance [?]. Discretization techniques can be classified according to several distinctions [?] [?] [?] [?]:

1. **Supervised *vs.* Unsupervised** [?]. Supervised discretization methods select cut points according to the class information while unsupervised methods discretize based on the attribute information only.

2. **Static *vs.* Dynamic** [?]. The number of intervals produced by many discretization algorithms is determined by a parameter k. Static discretization determines the value of k for each attribute while dynamic methods search for possible values of k for all features simultaneously.

3. **Global *vs.* Local** [?]. Most common discretization techniques convert attributes from continuous to discrete for all instances of the training

dataset, with only one set of discrete values for the entire learning process. That is known as *global discretization*. On the other hand, *local discretization* generates local sets of values (local regions) for a single attribute and apply each set at different classification context.

4. **Univariate *vs.* Multivariate** [?]. *Univariate* methods discretize each attribute in isolation, while *multivariate* discretization considers relationships among attributes.

5. **Eager *vs.* Lazy** [?]. *Eager* methods perform discretization as a pre-processing step, while *lazy* techniques delay discretization until classification time.

Figure 2.12 classifies six discretization methods where all the categories presented above are covered at least once.

Discretization Method	**Category**				
	1	2	3	4	5
Equal Width Disc. [?]	Unsupervised	Static	Global	Univar.	Eager
One-Rule Disc. [?]	Supervised	Dynamic	Global	Univar.	Eager
Entropy-Based Disc. [?]	Supervised	Dynamic	Local	Univar.	Eager
K-means clustering [?]	Unsupervised	Dynamic	Local	Univar.	Eager
Multivariate Disc. [?]	Supervised	Dynamic	Local	Multivar.	Eager
Lazy Disc. [?]	Unsupervised	Static	Local	Univar.	Lazy

Figure 2.12: Classification of various discretization approaches.

In the case of Naïve Bayes classifiers, discretization eliminates the assumption of normal distribution for continuous attributes. It forms a discrete attribute A'_k from a continuous one A_k. Each value $V'_{k,j}$ of the discrete attribute A'_k corresponds to an interval $(x_k, y_k]$ of A_k. If $V_{k,j} \in (x_k, y_k]$, $P(A_k = V_{k,j}|C_j)$ in Equation 2.9 is estimated by

$$\begin{aligned} P(A_k = V_{k,j}|Cj) &\approx P(a < A_k \leq b|C_j) \\ &\approx P(A'_k = V'_{k,j}|C_j). \end{aligned} \tag{2.13}$$

Since Naïve Bayes classifiers are probabilistic, discretization should result in accurate estimation of $P(C = C_j|A_k = V_{k,j})$ by substituting the discrete A'_k for the continuous A_k. Discretization should also be efficient in order to maintain the Naïve Bayes classifier low computational cost. Discretization performance is the main focus of this thesis as we search for an algorithm of linear complexity, but first, we present different discretization algorithms suited for Naïve Bayes Classifiers whose purpose is to improve classification accuracy.

2.3.3 Equal Width Discretization (EWD)

EWD [?] is one of the simplest discretization techniques. This unsupervised, global, univariate and eager process consists on reading a training set and for each attribute, sorting its values v from v_{min} to v_{max}, and generating k equally sized intervals, where k is a user-supplied parameter. Each interval has width $w = \frac{v_{max} - v_{min}}{k}$ and cut points at $v_{min} + iw$ where $i = 1, \ldots, k-1$. Let us now show how this works on the continuous attribute *temperature* in the non-discretized weather dataset [?] depicted in Figure 2.13.

Instance	*Attributes*				*Class*
	outlook	temperature	humidity	windy	play
1	sunny	85	85	false	no
2	sunny	80	90	true	no
3	overcast	83	86	false	yes
4	rainy	70	96	false	yes
5	rainy	68	80	false	yes
6	rainy	65	70	true	no
7	overcast	64	65	true	yes
8	sunny	72	95	false	no
9	sunny	69	70	false	yes
10	rainy	75	80	false	yes
11	sunny	75	70	true	yes
12	overcast	72	90	true	yes
13	overcast	81	75	false	yes
14	rainy	71	91	true	no

Figure 2.13: The weather dataset with some continuous attributes.

if $k = 7$ **then** $v_{min} = 64$ $v_{max} = 85$ **and** $w = \frac{85-64}{7} = 3$

Intervals	[64,67]	(67,70]	(70,73]	(73,76]	(76,79]	(79,82]	(82,85]
Temp.	64 65	68 69 70	71 72 72	75 75		80 81	83 85

Surprisingly, this basic scheme tends to significantly increase the performance of the Naïve Bayes Classifier [?] [?] even though it may be subject to information loss [**?**] and unstable partitioning among some other problems.

2.3.4 Equal Frequency Discretization (EFD)

EFD [?] seeks to divide the feature space into intervals containing approximately the same number of instances so partitions are more stable. This is accomplished by sorting the n instances of a feature space and grouping, into each of the k intervals, n/k adjacent (possibly identical) values. k is again a user defined parameter. As an example let us go back to the temperature attribute:

if $k = 7$ **then** $n = 14$ **and** $n/k = \frac{14}{7} = 2$

Temp.	64 65	68 69	70 71	72 72	75 75	80 81	83 85
Inst	2	2	2	2	2	2	2

The difference between EWD and EFD on classification accuracy is minimal in practice and both are widely used because of their simplicity.

2.3.5 1-R Discretization (1RD)

One-Rule learning [?], as explained before, is a very simplistic method but it is robust enough to handle both discrete and continuous data. Continuous values are discretized by a very straightforward technique. The idea behind this supervised discretization scheme is to divide the feature space into intervals so that each contains instances of one particular class. First, instances are sorted according to the value of the continuous attributes. This generates a sequence of class values that is later partitioned by placing cut points in the mid point between class changes. This procedure could generate many intervals, therefore a threshold *minbucketsize*, determining the minimum number of instances of the majority class in each interval (except the last one), is set. For example on the weather dataset of Figure 2.13:

if $minbucketsize = 3$

Temp.	64	65	68	69	70	71	72	72	75	75	80	81	83	85
Class	yes	no	yes	yes	yes	no	no	yes	yes	yes	no	yes	yes	no
Class	**yes**	no	**yes**	**yes**	yes	| no	no	**yes**	**yes**	**yes**	| **no**	yes	yes	**no**
Maj. Class			**yes**			|		**yes**			|		no	
Final	64	65	68	69	70	71	72	72	75	75	| 80	81	83	85

The attribute temperature went from fourteen values to just two: [64,77.5], (77.5,85]. Even though the attribute is initially partitioned into three values, if two consecutive intervals have the same majority class, then they can be merged into one with no adverse effect. Also, since this method discretizes from minimum to maximum (or left to right), the last bucket is formed even without complying with the *minbucketsize* parameter. The 1-R learner implements this discretization method *"on-line"* [?], meaning that it simultaneously learns and discretizes, although the same discretization practice could be performed as a preprocessing step or *"off-line"*.

2.3.6 Entropy Based Discretization (EBD)

First formulated by Fayyad and Irani [?], this discretization scheme was developed in the context of top-down induction of decision trees. In a similar way as C4.5 handles continuous attributes by setting cut points and evaluating the entropy measure, this discretization places a candidate cut point every midpoint between each successive pair of sorted values. For evaluating each cut point data is accommodated into two intervals and the resulting class information entropy is calculated. A binary discretization is determined by selecting the cut point with the lowest entropy measure among all candidates. This binary discretization is applied recursively until a stopping criterion - given by the *minimum description length principle*(MDL) - is met.

The *minimum description length* principle stops discretization if and only if the lowest entropy measure among all candidate points results in a split whose information gain is lower than a value resulting from the number of instances N, the number of classes c, the entropy of the instances E, the entropy of each of the two intervals E_1 and E_2, and the number of classes in each interval c_1 and c_2:

$$gain < \frac{\log_2 (N-1)}{N} + \frac{\log_2 (3^c - 2) - cE + c_1E_1 + c_2E_2}{N}$$

Since this algorithm was developed from a decision tree learning context, it tends to result in discrete attributes with small number of values. This is important for decision tree induction so it avoids the fragmentation problem [?].

Naïve Bayes classifiers do not suffer from this fragmentation issue, therefore forming small number of intervals may not be well justified for them.

Another incompatibility of the entropy-based discretization with Naïve Bayes classifiers is that measuring the information entropy of each attribute may reinforce the independence assumption of the Naïve Bayes classifiers by identifying cut points in the learning context of one-attribute applications instead of multi-attribute applications.

2.3.7 Proportional k-Interval Discretization (PKID)

PKID [**?**] is based on the equal frequency discretization policy, but focuses on adjusting discretization bias and variance by tuning the size of the intervals and, therefore, the number of them. Discretization bias is the discretization error that results from the use of a particular discretization policy. Discretization variance describes the discretization error that results from random variation of the data for a particular discretization algorithm. Variance measures how sensitive the discretization policy is to changes in the data. Discretization bias and variance are directly related to interval size and number. The larger the interval size, the smaller the interval number, the lower the variance, but the higher the bias. The opposite is also true: the smaller the interval size, the larger the interval number, the lower the bias, but the higher the variance [?].

The inverse relationship between interval size "s" and interval number "n" is trivial and may be represented by the formula:

$$s \times n = D. \tag{2.14}$$

Where "D" is the number of instances of the training set. The direct relationship between interval number and variance may be thought as the fewer the values an attribute has the less variation in the boundaries it is seen, therefore, more consistency is achieved. Finally, interval size and bias are directly related because the bigger the intervals formed by the discretization policy is, the more appropriate the interval becomes for random sampling.

Higher accuracy can be achieved by finding a good trade-off between the bias and variance, or what is the same, by tuning the interval size and number. PKID gives equal weight to discretization bias reduction and discretization variance reduction by setting the interval size equal to the interval number. Doing so requires:

$$s = n. \tag{2.15}$$

From Equation 2.14 and Equation 2.15 we can calculate interval size to be the square root of the total number of instances in the training dataset:

$$s = n = \sqrt{D}. \tag{2.16}$$

PKID sorts the attribute values and discretizes them into intervals of size proportional to the number of training instances. As the quantity of the training data increases, both discretization bias and variance may decrease making greater its capacity to take advantage of additional information.

One flaw of the PKID method is that for small training sets it forms intervals small in size which might not present enough data for reliable probability estimation, hence resulting in high variance and poorer performance of Naïve Bayes classifiers.

2.3.8 Non-Disjoint Discretization (NDD)

The idea behind NDD [?] is that calculating the probability estimation of a continuous value v_i that is discretized into an interval $(a_i, b_i]$ is more reliable if the v_i falls towards the middle of the interval instead of close to either a_i or b_i. Given s and n calculated as in Equation 2.16, NDD forms n' *atomic intervals* of the form $(a'_1, b'_1], (a'_2, b'_2], \ldots, (a'_n, b'_n]$ each of size s', where

$$\begin{aligned} s' &= \frac{s}{\alpha} \\ s' \times n' &= D \end{aligned} \tag{2.17}$$

Where α is any odd number and does not vary. For demonstration purposes let us say $\alpha = 3$.

With these atomic intervals in hand, when a value v is seen, it is assigned to the interval $(a'_{i-1}, b'_{i+1}]$ where i is the index of the atomic interval which contains v. Using this procedure v always falls towards the middle of the interval, except when $i = 1$ in which case v is assigned to the interval $(a'_1, b'_3]$, and when $i = t'$ in which case v is assigned to $(a'_{t'-2}, b'_{t'}]$

Grouping atomic intervals to form discretization values produces overlapping. That is the reason why this procedure is called non-disjoint.

2.3.9 Weighted Proportional k-Interval Discretization (WPKID)

A solution for the PKID problem of insufficient data in an interval was adopted in the WPKID algorithm. For smaller datasets, discretization variance reduction has a bigger impact on Naïve Bayes performance than discretization bias [**?**]. WPKID [?] weights discretization variance reduction more than bias for small training sets. This is accomplished by setting a minimum interval size m so the probability estimation always has a certain degree of reliability. When the training dataset is big enough to set the interval size above m (when $D = m^2$,) then discretization bias and variance are equally weighted as in PKID.

Going back to the equations for PKID, WPKID replaces Equation 2.15 for:

$$s - m = n$$

where $m = 30$ as it is commonly assumed the minimum sample space from which reliable statistical inferences should be drawn. This addition should alleviate the disadvantage of PKID for small datasets while keeping the advantages it carries for larger datasets.

2.3.10 Weighted Non-Disjoint Discretization (WNDD)

WNDD [?] is performed the same way as NDD, but with the restriction of the minimum interval size imposed by WPKID. As explained before, the minimum interval size parameter prevents the discretization algorithm from forming intervals with insufficient data for reliable probability estimation.

2.4 Summary

In this chapter we have presented a literature review consisting of two parts:

- First, we presented machine learning as a field followed by an explanation of the different techniques developed to accomplish data classification and eight classification algorithms.
- The second part deals with discretization and explains eight discretization algorithms from which six are suitable for the Naïve Bayes classifiers and two come from a decision tree induction context.

From the first section we can extract that the state-of-the-art classifiers C4.5 and Naïve Bayes have been well studied in literature and widely applied in practice. Overall, their classification performance is significantly higher than the rest. Although the classification accuracy obtained by C4.5 is sometimes better than Naïve Bayes, the computational complexity of Naïve Bayes makes it ideal for incremental learning and for standard learning on larger datasets. It only requires one pass through the data, which results on a linear time complexity, and does not store the instances of the dataset; only counters, resulting in a space complexity that is not bounded to the size of the dataset. All other classifiers serve as benchmarks for classification performance of new methods.

From the discretization section we can highlight the importance of discretization in the context of classification. It serves four purposes:

1. Allows learning on continuous datasets for discrete space classifiers.
2. May improve classification accuracy.
3. Increases the classifier's performance time and memory wise.
4. Decreases the complexity of the learned theory making it simpler and more understandable.

Although some of the discussed discretization algorithms perform well on small to medium size datasets, only one, EWD, can be performed in linear time*, the remaining ones require sorting, therefore augmenting their time complexity. Added to that, all of them, except EWD, require storing all the training instances, thus incurring in a linear (or wider) space complexity. These two characteristics make the discretization techniques studied unsuitable for incremental learning and do not match the time and space complexity inherent to the Naïve Bayes classifier.

In Chapter 4, we will introduce our own incremental discretization algorithm that performs within the computational complexity of the Naïve Bayes classifier and still serves the purposes described above.

* EWD is linear in time, or takes $O(n)$ where n is the size of the training dataset, but it needs to reach the end of the dataset before a second pass for the actual data conversion therefore it is unsuitable for infinite datasets

Chapter 3

Experimental Procedures for Assesing Classification Algorithms

3.1 Introduction

Estimating the accuracy of a classifier induced by supervised learning is important not only to predict its future prediction accuracy, but also for choosing a classifier from a given set, or combining classifiers for a particular task [?] [?].

In this chapter we offer a study of the most widely used techniques for the proper performance evaluation of learners and the comparison between them. Later on, we propose two testing platforms used throughout the remainder of this thesis, we explain the strengths of these techniques and how they are implemented.

3.1.1 Background

The performance of a classifier is usually measured in terms of either the *error rate* or the *success rate*, also called *accuracy*. The classifier's job is to predict the class of each instance, and if the prediction matches the actual value, that is, if the prediction is correct, then it is counted as a *success*, if not, it is an *error*. The error rate is the proportion of errors over the whole set of classified instances. Similarly, the accuracy, or success rate, is the ratio of successes over the number of classified instances. The error- and success-rate add up to 1, so one can be easily calculated from the other. We prefer to measure the performance in terms of accuracy to measure the overall performance of the classifier.

Performance estimation can be partitioned into a *bias*, a *variance* and an *irreducible* term [?] [?] [?] [?] [?] [?]:

Bias: The bias of a method designed to estimate a parameter is defined as the expected value minus the estimated value. It describes the component of performance that results from systematic error of the learning algorithm.

Variance: The variance measures the sensitivity of the assessed algorithm to changes in the training data. It describes the component of performance that results from random variation in the training data and from random behavior in the learning algorithm. As an algorithm becomes more sensitive, its variance increases.

Irreducible term: The irreducible term describes the performance of an optimal algorithm. It is usually aggregated with the variance and the bias terms resulting in the true performance measure.

Given that the irreducible term is fixed and the variance and bias are variable, we can infer that there is a trade-off between bias and variance. That is, all other things being equal, if the algorithm in question is modified, it will have opposite effects on the bias and variance. Even though the best estimation is the one that minimizes both, bias and variance, low variance is preferred [?].

Since we want to forecast the performance of the learner in future data, testing on the training data is not a good predictor for performance on new data. It adjusts the classification theory up to the point that it *over-fits* the particular dataset, not saying much about the performance on unseen data. This kind of testing results in what is know as *resubstitution error* which is an optimistic performance evaluator. Assessing the classifier's performance on future data requires a test dataset that has played no part in the formation of the predictive model. This independent dataset is called the *test set* and is assumed to be representative of the underlying problem.

Generally, a dataset is provided to generate a classification theory. On rare occasions an independent test set is given. One technique to evaluate the performance of the classifier on new data is known as *holdout*. This technique splits the dataset in two disjoint subsets, one subset is the training set and the other one is the test, or holdout set. Usually, one-third of the data is designated for testing and the remaining two-thirds for training. The classification algorithm learns from the training set and then evaluates its theory on the test set resulting in a single accuracy estimate. This value is an approximation to the true accuracy of the classification theory on unseen data, but we need to estimate how close they are from one another. Statistics tells us that the bigger the test set the smaller the confidence interval the approximation lies within. That means that the variance of the assessment is reduced and the resulting accuracy estimate is a better approximation to the *true* one. The problem arises when the given dataset is not large enough to give us a comfortable confidence. In this case, a very common case in *classic* data mining, it is required to come up with more than one accuracy estimate from different partitions of the dataset. Various techniques to accomplish this have been developed in the past.

3.2 Learner Evaluation

First we introduce the common testing procedure known as “cross-validation”, then we explain why this method is insufficient and how it can be modified to more adequately avoid testing bias. Next, we discuss significance test and other statistical methods necessary for the proper comparison between two (or more) algorithms.

3.2.1 Cross-Validation

We want to use as much of the data as possible for training in order to get a good classifier, but at the same time we want to test as much of it as possible to get a good accuracy estimate. *Cross-validation* is one method widely used to deal with this problem [**?**] [**?**] [?] and our preferred method to asses overall learner performance. It consists of dividing a dataset into equal sized fractions (e.g. three one-thirds) called *folds* and sequentially holding out one fold for testing and the remaining ones for training until all subsets have served for both training and testing purposes. Even class representation is required for both the training and the test sets, therefore we need to ensure that each one of the folds contains approximately the same class distribution as the whole dataset. This is known as *stratified* n*-fold cross-validation* where n is the number of (training set / test set) pairs. Learning on each one of these folds results in n usually different accuracies that are then averaged to give a better approximation to the true accuracy estimate. The pseudo-code for the standard stratified n-fold cross-validation is shown in Figure 3.1

Tenfold cross-validation is a standard method used for measuring the performance of a learning algorithm. In it, the original data set is divided into ten disjoint groups, each one with approximately the same class distribution.

```
1.  Divide dataset $D$ into $n$ approximately equal folds $d_i$
2.  **for** each $d_i$ {
3.      $Acc_i \leftarrow$ Train on $D - d_i$ and Classify on $d_i$
4.  }
5.  $MeanAcc \leftarrow \frac{1}{n} \sum_{i=1}^{n} Acc_i$
6.  **return** $MeanAcc$
```

Figure 3.1: n-fold cross-validation pseudo-code

Each subset is then used to form a test set for training on the remaining nine, i.e. Subset one is used for testing and subset two through ten are used for training, then subset two is the testing subset and subsets one, and three through ten form the training set. This is then repeated for each subset. This way we come up with ten different accuracy measures where data used for training is different from the one used for testing. These ten results are then averaged and the mean accuracy is the final measurement of classification performance.

3.2.2 Ten by Ten-Fold Cross-Validation Procedure

According to Boucaert in [?], and Witten and Frank in [?], estimating accuracy from ten samples is not enough. It is necessary to take more examples into account and randomize the process to avoid bias. One approach we use in this thesis is known as $m \times n$-fold cross-validation, or $m \times n$FCV. It consists of repeating the n-fold cross-validation process, on the same data set, m times while randomizing the order of the data at each iteration as shown in the pseudo-code in Figure 3.2. Statistically, it is a better estimation of the accuracy of the classifier in the new data because it is drawn from the bigger sample size $m \times n$.

```
1. **repeat** $m$ times {
2.     $D_m \leftarrow$ all instances of data set $D$ in random order
3.     $Acc_m \leftarrow n$-fold cross-validation($D_m$) as in Figure 3.1
4. }
5. $MeanAcc \leftarrow \frac{1}{m}\sum_{i=1}^{m} Acc_j$
6. **return** $MeanAcc$
```

Figure 3.2: $m \times n$-fold cross-validation pseudo-code

Several issues arise with the introduction of this scheme. The first one is how to estimate the accuracy from the set of results returned. Another matter is the statistical comparison between estimations.

These issues are analyzed in Boucaert's paper [?]. In it, Boucaert recommends the use of all 100 individual estimates to calculate the mean accuracy and its variance. Then, when comparing these estimates coming from two different algorithms, we compare them by performing a t-test with 10 degrees of freedom. This simple setting, known as the *use-all-data* approach, is calibrated to compensate for the difference between the desired Type I error and the true one and has the same properties of more complicated tests. We use this method within our own test procedure as explained in the next section.

3.3 Comparing Learning Techniques

Comparing two learning algorithms is not a simple task [?]. In an effort to compare the results coming from different machine learners, we have developed a test platform that performs "ten by ten-fold Cross-Validation" (10x10FCV). It allows us to compare various classifiers against each other while assuring that comparisons are statistically significant.

3.3.1 The Test Scenario

For the first testing scheme we use a script that does the following on each dataset D:

1. Splits the dataset into ten disjoint subsets with equal class representation ($D = D_1 \cup D_2 \cup \cdots \cup D_{10}$).

2. Generates ten training/test-dataset pairs. For each iteration i the test set is D_i and the training set is the union of the remaining nine subsets.

3. Runs each training/test-dataset pair on each of the competing algorithms.

4. Reports the accuracy of all the runs to a file. One file per algorithm.

5. Calculates the average mean and standard deviation on the results in each file according to the *use all data* technique previously described.

6. Performs t-test statistics to compare the results of the classifiers with a significance level of 0.05 or 95% confidence and 9 degrees of freedom.

7. Generates a win-loss table according to the outcomes of the test statistics.

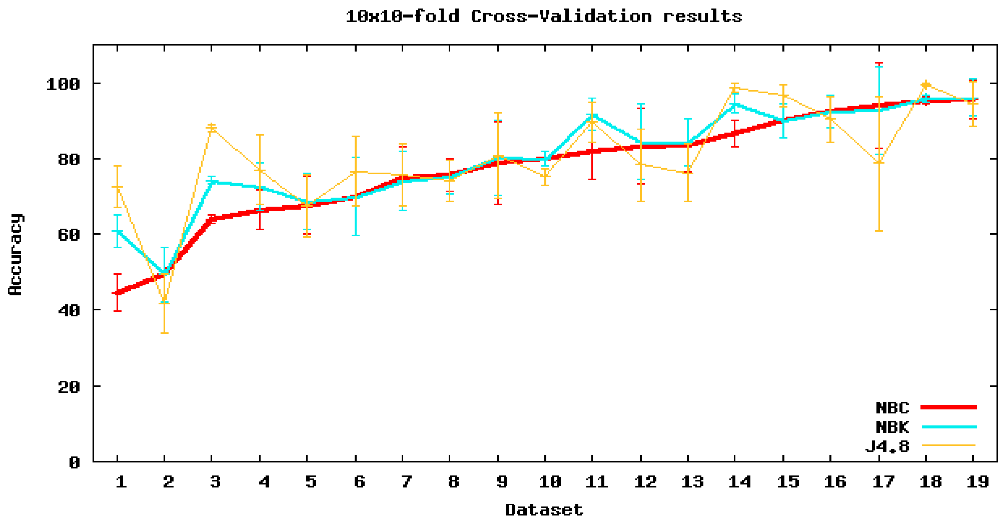
8. Graphs the mean accuracy and +/− one standard deviation.

As Boucaert's paper suggests, this testing scenario is robust to *Type I errors* and accommodates to the requirements established for a proper performance evaluation setting (high replicability, low variance). An example of this set-up is presented next.

3.3.2 An example: NBC Vs. NBK and C4.5

The example in Figure 3.3 shows the performance of NBC, NBK and J4.8 on nineteen datasets from the UCI repository [?].

The graph displays the accuracy of each classifier on each dataset sorted in ascendant order in reference to the NBC results. From this graph we can ratify Wolpter's claim that states, "...with certain assumptions, no classifier is always better than another one" [**?**]. It is not clear that the tree induction classifier C4.5 (Weka's J4.8) is better than NBC and NBK in terms of accuracy, as their curves often intercept in the graph. That is why the win-loss table is also provided. Each row shows the number of times the algorithm performs significantly better (*wins*) or worse (*losses*) than the accuracy of the other algorithms. The $win - loss$ column determines which



#	Dataset	Insts	Atts	Num	Disc	Cl
1	vehicle	846	18	18	0	4
2	primary-tumor	339	17	0	17	21
3	letter	20000	16	16	0	26
4	vowel	990	13	10	3	11
5	horse-colic	299	24	8	16	3
6	audiology	226	69	0	69	24
7	auto-mpg	398	6	6	0	4
8	diabetes	768	8	8	0	2
9	echo	130	8	6	2	2
10	waveform-5000	5000	40	40	0	3
11	ionosphere	351	34	34	0	2
12	hepatitis	155	19	6	13	2
13	heart-c	303	13	6	7	2
14	anneal	898	38	6	32	5
15	vote	435	16	0	16	2
16	soybean	683	35	0	35	19
17	labor	57	16	8	8	2
18	hypothyroid	3772	29	7	22	4
19	iris	150	4	4	0	3

Alg	w-l	win	lose
j48	5	15	10
nbk	3	10	7
nbc	-8	5	13

Figure 3.3: Comparison between Naïve Bayes classifier, with and without kernel estimation (NBC and NBK respectively), and Weka's implementation of C4.5 (J4.8) using a stratified 10x10FCV technique. The graph plots the mean accuracy and +/- one standard deviation of each dataset. The table summarizes the characteristics of the datasets used for this comparison. Finally, the win-loss table gives us a higher level view on the performance of the classification techniques.

algorithm achieves the overall highest accuracy of the compared algorithms on the tested datasets. That does not necessarily mean that such algorithm is the preferred one. Perhaps we would need a very reliable algorithm. In this case, we would choose the one that looses the least, or maybe we would prefer to risk a little for a better performance and choose the one that would win the most.

3.4 Learning Curves

In this section, we present an additional test scenario where the proportion of training instances over the test instances varies for each run. Although it is a very time consuming and memory expensive approach, we propose it for comparing the learning curves of different algorithms on individual datasets. This procedure assesses the accuracy of classifiers at different quantities of training data generating a learning curve proportional to the amount of training. We named it *incremental 10-times 10-fold cross-validation* and it is explained next.

3.4.1 Incremental 10-times 10-fold Cross-Validation

This evaluation procedure performs 10x10FCV on increments of ten percent of training data. That is, it runs over train sets containing ten percent of the data, and tests on the remaining nine sets. Then it increments the training data to twenty percent and repeats the randomization procedure. It repeats this process nine times until the last set of runs are equivalent to the 10x10FCV technique explained above.

A simple but powerful implementation of this method was developed by Dr. Menzies in a software package called “bill”. It is part of a series of data mining tools in a linux environment[*] that can be downloaded from www.scant.org. “Sequence” is the particular program used within this package to assess the learning curves of different algorithms on different datasets.

Sequence takes a dataset and splits it into n stratified and equally sized subsets $D_i \ldots D_n$. Then it trains each specified algorithm on subset D_i and tests on the remaining subsets repeating this procedure n times until all the subsets have served as training set. In every successive iteration j, sequence adds the subset D_{i+j} to the training set and therefore subtracts it from the test set. It iterates until there is only one subset left for testing (until $j = i - 1$). Finally, it repeats the whole process m times while randomizing the order of the original dataset.

Since we set *sequence*'s parameters to $m = 10$ and $n = 10$, it runs each algorithm a total of 900 times on each dataset. It generates 9 different 10-fold cross-validations, one per each 10% extra training data. This is a very complete test that shows the performance of each algorithm at different training states. It also tells us how fast (or slow) an algorithm can improve its classification accuracy. This is most important in the context of model

[*]We use **cygwin** for all testing and development. It is a linux emulator under windows and it is downloadable from www.cygwin.com

selection for a particular application. Next we will see an example of this test scenario at work.

3.4.2 Example

Figure 3.4 illustrates the learning curve of NBC, and J4.8 on ten UCI datasets [?]. These graphs help us visualize how fast and how accurate the learners may be on certain datasets.

Notice the flattened shape of the majority of the learning curves. Perhaps it indicates a stabilization of the predictive model in the early training stages. This is exactly what we found after experimental tests on several datasets. We exploit this “early plateauing” of the majority of studied datasets when developing our classification tool in Chapter 5.

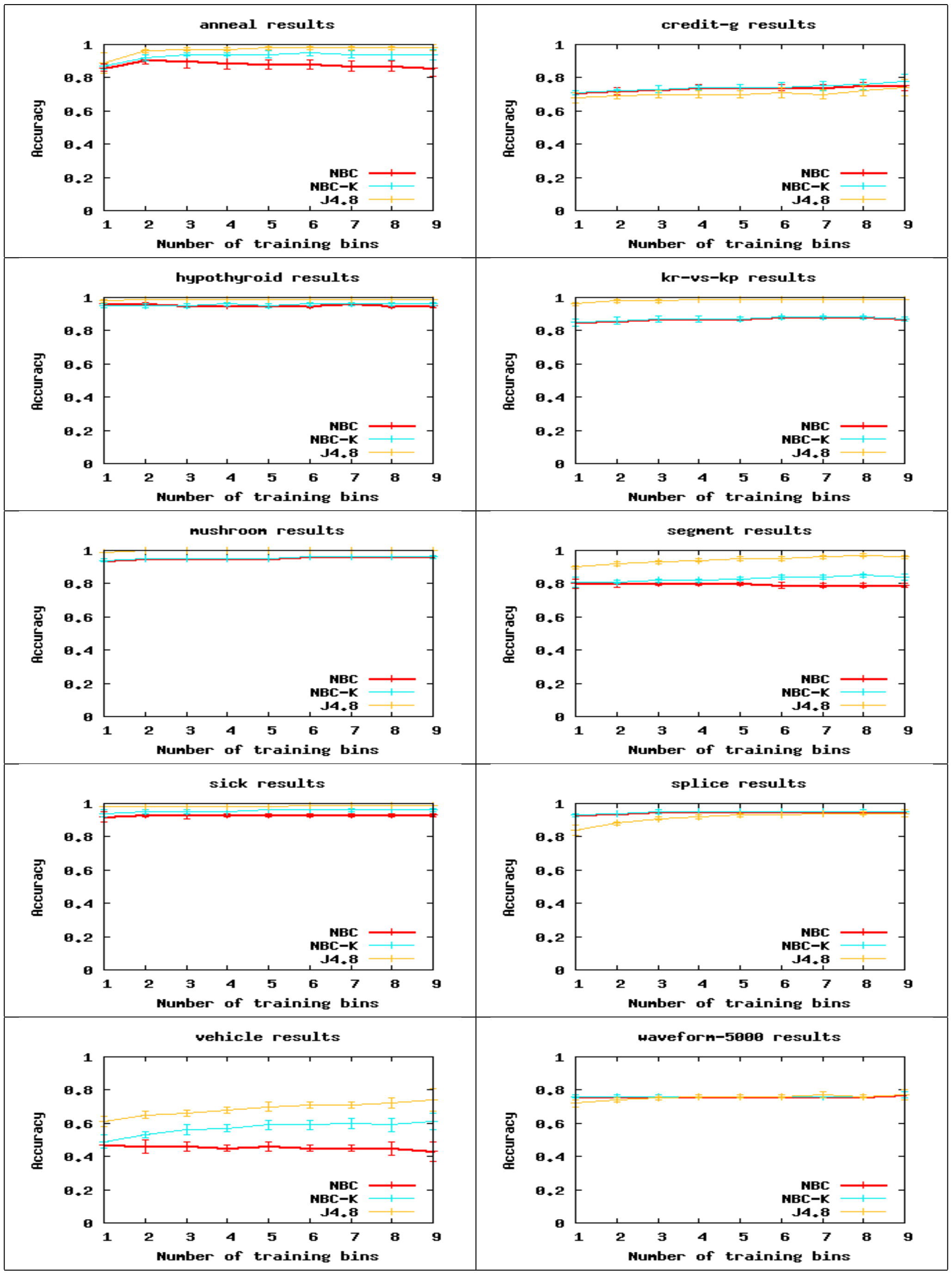


Figure 3.4: Incremental performance comparison between NBC and C4.5.

Chapter 4

SPADE: Single Pass Dynamic Enumeration

4.1 Introduction

Many discretization techniques have been developed to improve the classification accuracy of the Naïve Bayes classifier. While some of them have achieved great enhancement, like the WNDD (§2.3.10), none of them have been able to scale parallel to the classifier. That is, no algorithm can discretize continuous attributes as they come, reading them once, and without storing them. The challenge of building a discretization process that matches the characteristics of the Naïve Bayes classifier while improving classification accuracy and minimizing the lost of information, motivated us to invent SPADE, *S*ingle *PA*ss *D*ynamic *E*numeration, as described in this chapter.

The ultimate goal of our discretization technique is to provide the most accurate density estimate while updating the partition boundaries incrementally and minimizing information loss.

4.1.1 Creating Partitions

The idea behind SPADE is to create dynamic “ranges”, also called “buckets”, “bands”, or “bins”, that self adjust every time a new continuous value is encountered. We achieve this by assigning to the first value, v1, a band of the form b1=[v1, v1] and min=v1, and max=v1. When the second value, v2, comes by, it is tested to see whether it is greater than the maximum value seen so far (“max”=v1), lesser than the minimum value seen so far (“min”=v1), or within “min” and “max”. Depending on the result of the test, three things can occur:

- If v2 lies between min and max (v2=v1 in this case), then we look for the band that contains v2 (in this case b1) and return it to the learner.
- If v2 is greater than max, then b2=(max , v2] is created and returned to the learner. Therefore the bands are now b1=[v1 , v1], b2=(v1 , v2], and max=v2.
- If v2 is less than min, then b2=[v2 , v2] is created and returned, while b1 is modified from [v1 , v1] to b1=(v2 , v1]. This procedure leaves the bands b2=[v2 , v2], b1=(v2 , v1], and min is updated to min=v2.

This procedure is repeated for every continuous value of each attribute.

Even though this is a single pass approach (each continuous value is read once), it has several problems. One serious problem is that if the initial values are the minimum and maximum of the attribute's distribution, then only two bands will be created, and most of the values will fall into only one band. This defies the purpose of the discretization since all class-value information is lost [?] [?]. Another problem arises if the continuous values come in ascending or descending order. In either case, one band is created for each value and this also cancels out the point for discretization [?].

In order to overcome these problems two improvements are necessary. Solving the first issue requires creating a minimum number of bands to guarantee a minimum continuous attribute partitioning. The other problem, great amounts of bins, is solved by updating the bands often. Both techniques are explained next.

4.1.2 Band Sub-Division

A new parameter is introduced to the algorithm, "SUBBINS". This tells the discretizer how many equal-width sub-divisions to create per each band generated by the original algorithm. In other words, each time a band is created, it is split in SUBBINS number of (sub-)bands. This approach assures a better distribution among various bands, guarantees a minimum number of them, and avoids crowding all values into a single band.

If we go back to the example in the previous section and assume that

SUBBINS=3, then:

- If v2 lies between min and max (v2=v1 in this case), then we look for the band that contains v2 (in this case b1) and return it to the learner.
- If v2 is greater than max, then b2=(max , v2] is partitioned in three so b2=(max , max+(v2-max/3)], b3=(max+(v2-max/3) , max+2(v2-max/3)], b4=(max+2(v2-max/3) , v2] are created and b4 is returned to the learner. Therefore the bands are now b1=[v1 , v1], b2=(max , max+(v2-max/3)], b3=(max+(v2-max/3) , max+2(v2-max/3)], b4=(max+2(v2-max/3) , v2] guaranteeing at least four bands.
- If v2 is less than min, then b1 is modified from [v1 , v2] to b1=(min+2(min-v2/3) , v1], and b2=(min+(min-v2/3) , min+2(min-v2/3)], b3=(v2 , min+(v1-v2/3)], b4=[v2, v2] are created and b4 is returned. This procedure leaves the bands b4=[v2 , v2], b3=(v2 , min+(v1-v2/3)], b2=(min+(min-v2/3) , min+2(min-v2/3)], b1=(min+2(min-v2/3) , v1] again, assuring at least four bands are created.

With this technique, some control over the number of bands created is gained. Also, since there is no partitioning after instances have been assigned to each band, there is no information loss due to the split. Now the problem is that a big number of SUBBINS can potentially generate a great number of unnecessary bands. To solve this new problem along with the previous issue, creating one band per continuous value when values are sorted, the following technique has been developed.

4.1.3 Updating Partitions

Controlling a variable number of bands allowed to be maintained by the learner is necessary. Creating huge amounts of bands and searching them is exceedingly time consuming. Also, they might not contain enough values to sample from for a good estimator. In order to gain such control, several new parameters are introduced to SPADE. They help direct a "merge" function whose job is to join unused or under-used bands; bands with small number of instances.

Three parameters are introduced to the algorithm:

MAXBANDS: A dynamic upper-bound for the number of bands. It is set to the squared root of the number of training instances seen.

MAXINST: A dynamic upper-bound suggesting the maximum number of instances in each band (suggested upper-bound tally). It is set to two times MAXBANDS.

MININST: A dynamic lower-bound suggesting the minimum number of instances in each band (suggested lower-bound tally). It is set to the same number as MAXBANDS

We have chosen the square root of the processed training instances as SPADE's border criterion to control discretization bias and variance as explained by Yang and Webb in [?]. Also, we called MAXINST and MININST "suggested" boundaries because there is no guarantee to hold the number of instances in a band within those values. They simply prevent the fusion of any bin with one having a tally bigger than MAXINST, and promote the combination of multiple sequential under-MININST bins until their sum surpasses the MININST parameter.

The bin merging function works as follows. The number of bands created for each attribute is maintained and if it surpasses MAXBANDS, then the merge function is called. This function visits each one of the bands in the current attribute sequentially. If the visited band has fewer instances than MININST, and if when combined with the next it is still less than MAXINST, the two bands are collapsed into one. All other counts have to be updated accordingly. If any of the tests fail, then the current band is not merged and the function continues operation on the next band. A clearer view of this procedure is offered in the next chapter, where the whole learner is explained.

Merging bands according to the above procedure minimizes the information loss due to discretization because all counts are actual counts and no estimation is used for any operation.

4.1.4 Algorithmic Complexity

A comparison of the algorithmic complexity of Naïve Bayes with the single Gaussian assumption, John and Langley's kernel estimation, and SPADE is shown in Figure 4.1. Notice that there is no difference between the time complexity of NBC and NBK on training data, and SPADE is actually worse than them in this context by the b factor. Experimental procedures show

Operation	Gaussian Assumption		Kernel Estimation		SPADE	
	Time	Space	Time	Space	Time	Space
Train on n instances	$O(nk)$	$O(k)$	$O(nk)$	$O(nk)$	$O(nk \ln b)$	$O(kb)$
Test on m instances	$O(mk)$		$O(mnk)$		$O(mk \ln b)$	

Figure 4.1: Algorithmic complexity of three different continuous handling techniques for the Naïve Bayes classifier, given k attributes. Also, b is the number of bins generated by SPADE.

that b very rarely reaches the worst case or even close to the worst case. We describe the very unlikely scenario where it would get to the worst time and memory case in §5.4.4. What algorithm complexity does not show is what we call the "two-scan" problem.

Many kernel estimation and discretization methods violate the *one scan* requirement of a data miner; i.e. learning needs only one scan (or less) of the data since there may not be time or memory to go back and look at a store of past instances. For example, the EWD discretization method is *n-bins* which divides attribute a_i into bins of size $\frac{MAX(a_i)-MIN(a_i)}{n}$. If MAX and MIN are calculated incrementally along a stream of data, then each instance may have to be cached and re-discretized if the bin sizes change. An alternative is to calculate MAX and MIN after seeing *all* the data. Both cases require two scans through the data, with the second scan doing the actual binning. Many other discretization methods we discussed in §2.3 and all the methods discussed by Dougherty et.al. [?] and Yang and Webb [?], suffer from this two-scan problem. Similarly, John and Langley's kernel estimation method cannot build its distribution until *after* seeing and storing all the continuous data.

SPADE only scans the input data once and, at anytime during the processing of X instances, SPADE's bins are available. Further, if it ever adjusts bins (e.g. when merging), then the information used for that merging comes from the bins themselves, and not some second scan of the instances. Hence, it can be used for the incremental processing of very large data sets.

Going back to the discretization classification shown in Figure 2.12, SPADE is classified as an **unsupervised, dynamic, local, univariate, eager** discretization technique. It is **unsupervised** because bins are created and merged according to individual values and not the instance's class. It is **dynamic** in the sense that there is no predefined number of bins SPADE creates, but this number dynamically changes as data comes in. Also, since discretization takes place during training time, and the bins generated are not the same for the whole training set, this discretization technique is **local**. SPADE is **univariate** as it discretizes each continuous attribute value independently, and attribute correlations are not taken into account for the process. It is **eager** because it does not wait until classification time to generate the bins. Finally, and more importantly, SPADE is defined as a **one-pass**, **on-line**, **incremental** discretization algorithm.

4.2 Experimental Results

Figure 4.2 and Figure 4.3 show the results of our first experiment. In it, we compare the histogram of the true distribution of the data and the estimated density distribution generated by SPADE.

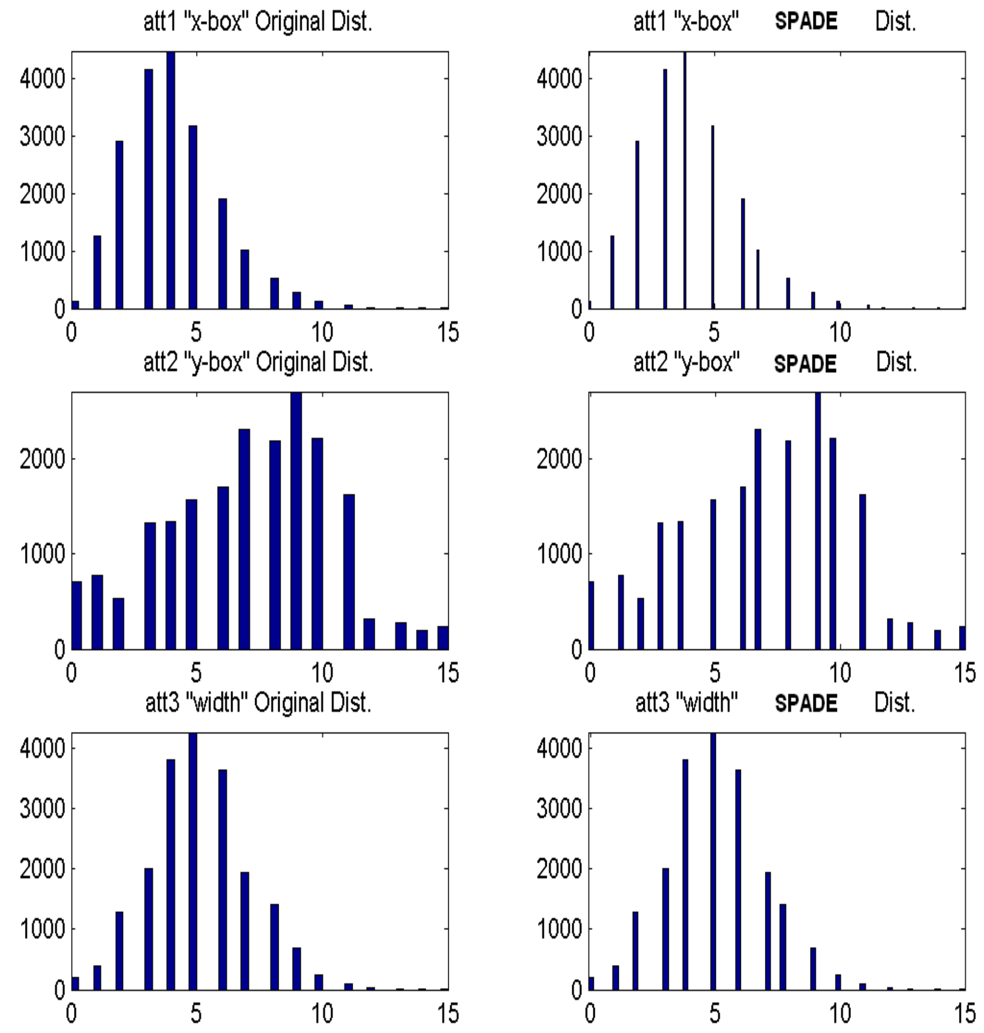


Figure 4.2: First three continuous attributes from the UCI dataset “letter”.

The first six attributes of the “letter” dataset from the UCI Repository [?]

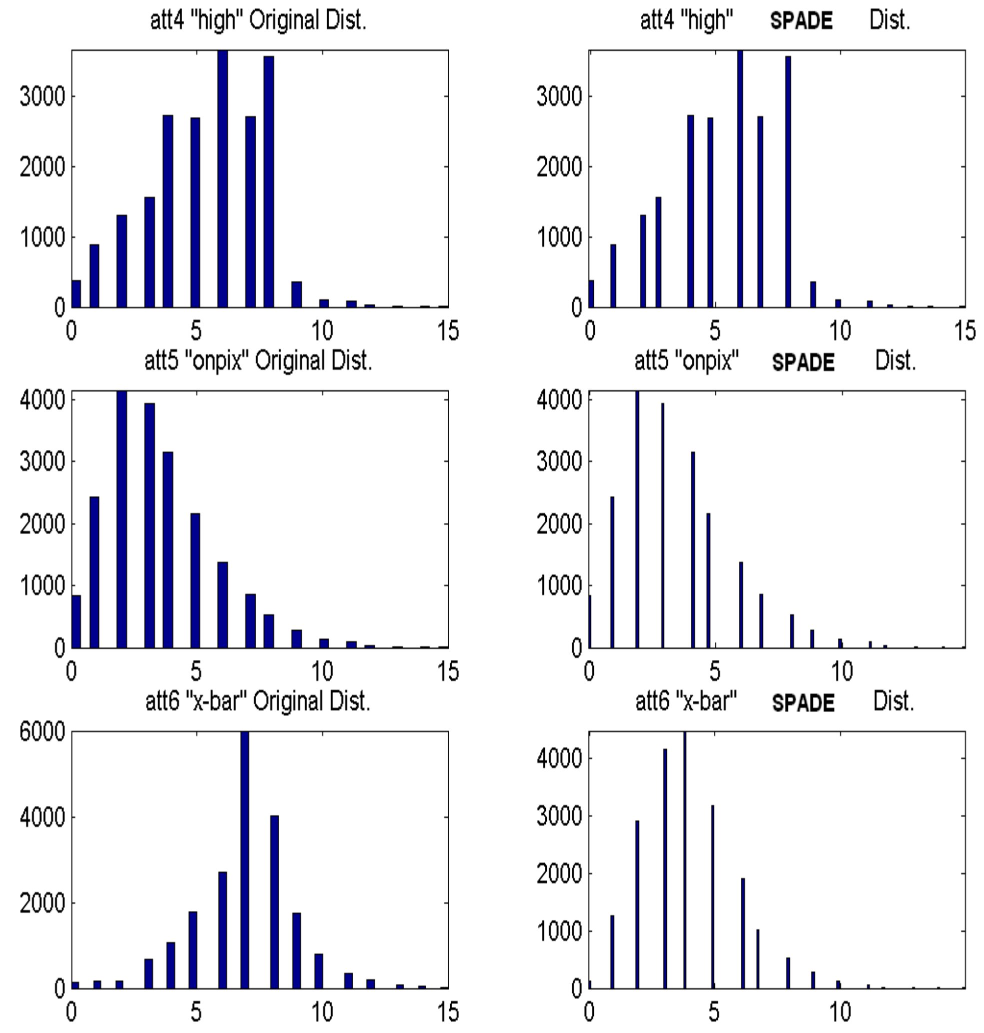
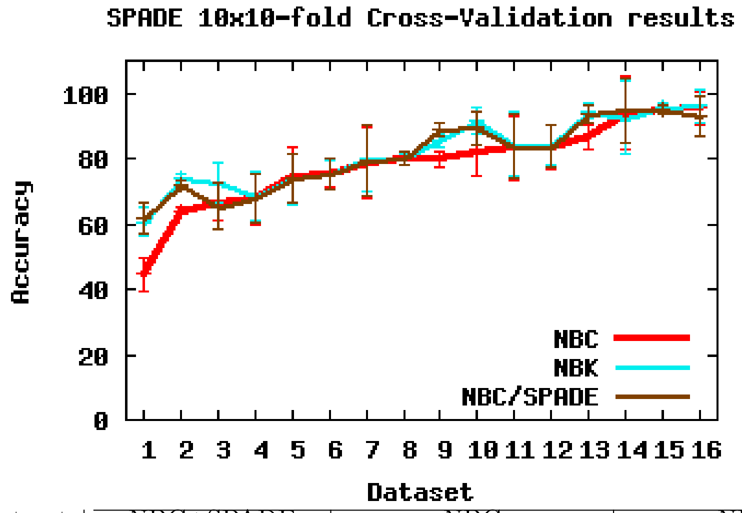


Figure 4.3: Attributes four to six from the UCI data set “letter”.

were used for this comparison. We use Matlab 6.2 to generate the histograms from the raw data, and then we let SPADE run on the dataset and generate its own bins. Next, we plot the tallies of each bin generated. Notice from the graphs, that there is no major difference between the corresponding graphs. SPADE's density estimation is very close to the true one, even though they do not always follow a Gaussian distribution. The minimum variation between the graphs is expected and it is a consequence of the loss of information generated by going from a fine grain level of measurement to a coarser one (See §2.3.1). In summary, from these results, we can say that SPADE's loss of information is minimal. Therefore, one of the discretization goals introduced at the beginning of this chapter has been attained.

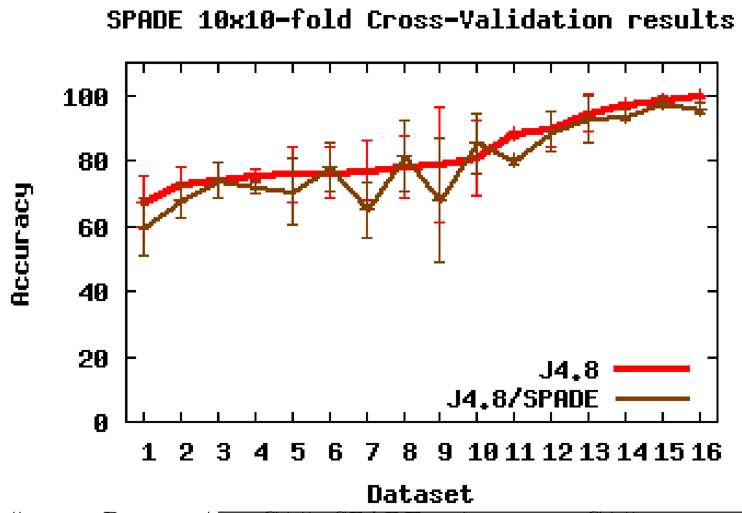
The second test's objective is to see the difference SPADE makes on existing classifiers. Figure 4.4 shows the results of comparing Naïve Bayes with SPADE (NBC+SPDE), a single Gaussian (NBC), and kernel estimation (NBK). As we expected, Spade improves the performance of the Naïve Bayes classifier as it was developed in the context of Bayesian classifiers like EWD, EFD, PKID, (§2.3). In the figure, the table displays the accuracies of NBC+SPADE, NBC and NBK, as well as the statistical comparison between NBC+SPADE and the each of the other two. From that table, notice that SPADE significantly improves NBC's accuracy five times and decreases it once. Also the average accuracy across all datasets is increased very close to the one obtained by NBK. This is encouraging as it demonstrates that in most cases SPADE is at least as accurate as NBC and in some cases it is actually better.



#	Dataset	NBC+SPADE	NBC	NBK
1	vehicle	61.83375± 4.86358	44.64230± 4.95235 -	60.76793± 4.28329
2	letter	72.21600± 1.14537	64.04600± 1.11137 -	74.27700± 1.08412 +
3	vowel	65.41417± 7.19013	66.42427± 5.30648	72.55558± 6.33344 +
4	horse-colic	68.01264± 7.63013	67.57586± 7.63559	68.81494± 7.44340
5	auto-mpg	74.18719± 7.55786	74.85193± 8.36961	73.94808± 7.82594
6	diabetes	75.28351± 4.71252	75.54479± 4.35598	75.10050± 4.63620
7	echo	79.30771±10.91667	78.84617±11.00827	80.23079± 9.97605
8	waveform-500	80.14200± 1.79838	80.00200± 1.92852	79.84400± 1.94834
9	segment	88.96105± 2.14164	80.04331± 2.33934 -	85.73162± 2.08922 -
10	ionosphere	89.45317± 5.07635	82.09206± 7.69688 -	91.73413± 4.30545 +
11	hepatitis	83.80833±10.00035	83.27917±10.03757	84.37500± 9.91803
12	heart-c	83.67312± 6.49099	83.50108± 6.97044	84.16345± 6.23720
13	anneal	93.73146± 2.98953	86.60624± 3.59273 -	94.48851± 2.47123
14	labor	94.90000± 9.77244	93.86666±11.25882	92.66666±11.45920
15	hypothyroid	95.30524± 1.31992	95.30240± 1.06445	95.92538± 0.98183
16	iris	93.19999± 5.99138	95.66666± 5.13564 +	96.13332± 4.84832 +
	MEAN	81.21433	78.26818	81.92231

Figure 4.4: Comparison between Naïve Bayes with data preprocessed by SPADE, NBC (Gaussian), and NBK (kernel estimation). The "+" and "-" signs indicate that there is a statitistical significant increase or decrease in accuracy of the method against SPADE respectively.

On the other hand, Figure 4.5 shows the effects of SPADE on C4.5. According to Yang [?], supervised discretization performs better in decision trees. On the other hand, unsupervised techniques, like SPADE, tend to do worse in this context because they do not partition the attributes separating the different classes. This is the reason why we do not expect SPADE to improve C4.5 accuracy.



#	Dataset	C4.5+SPADE	C4.5
1	horse-colic	67.31494 ± 8.17739	59.88391 ± 8.77245 -
2	vehicle	72.42493 ± 5.52383	67.69398 ± 5.28800 -
3	diabetes	74.15074 ± 5.41005	73.90228 ± 5.23319
4	waveform-500	75.12000 ± 2.03107	72.12400 ± 2.05849 -
5	auto-mpg	75.73270 ± 8.17346	70.66411 ±10.32592 -
6	heart-c	76.22581 ± 7.70545	78.10538 ± 7.39768
7	vowel	76.90911 ± 9.14462	65.14144 ± 8.52095 -
8	hepatitis	78.27500 ± 9.48724	81.37083 ±10.70506 +
9	labor	78.66667 ±17.65106	67.80000 ±19.08819 -
10	echo	80.76925 ±11.33595	85.38463 ± 9.24499 +
11	letter	88.01050 ± 0.83416	79.99550 ± 1.05009 -
12	ionosphere	89.65873 ± 5.28187	89.03968 ± 6.02286
13	iris	94.53333 ± 5.87027	92.86666 ± 7.23223
14	segment	96.86580 ± 1.19171	93.95672 ± 1.63248 -
15	anneal	98.68553 ± 1.01912	97.88478 ± 1.48782
16	hypothyroid	99.53342 ± 0.31996	95.97325 ± 1.77369 -
	MEAN	82.67978	79.4867

Figure 4.5: Comparison between C4.5 with data preprocessed by SPADE, and C4.5 with the continuous data. The "+" and "-" signs indicate that there is a statitistical significant increase or decrease in accuracy of C4.5 with continuous values against C4.5 with SPADE respectively.

From Figure 4.5 and its table, it is clear that SPADE could decrease the accuracy of the learner. This is neither surprising nor discouraging, as it is an incremental discretizer designed to work specifically in the context of Bayesian learners. It also proves that discretization algorithms ought to be targeted to a specific learning approach, and not all discretization techniques work on all learners.

Lastly, Figure 4.6 summarizes the results from Figure 4.4 by comparing the results from SPADE and John and Langley's kernel estimation method using the display format proposed by Dougherty, Kohavi and Sahami [?]. In that figure, a 10x10-way cross validation used three learners: (a) Naïve Bayes with a single Gaussian; (b) Naïve Bayes with John and Langley's kernel estimation method (c) Naïve Bayes classifier using data pre-discretized by SPADE. Mean classification accuracies were collected and shown in Figure 4.6, sorted by the means $(c - a) - (b - a)$; that is, by the difference in the improvement seen in SPADE *or* kernel estimation *over or above* a simple single Gaussian scheme. Hence, kernel estimation works comparatively better than SPADE on datasets A through I, while SPADE performs comparatively better on the remaining six datasets (J, K, L, M, N, O).

Three features of Figure 4.6 are noteworthy. Firstly, in a finding consistent with those of Dougherty et.al. [?], discretization can sometimes dramatically improve classification the accuracy of a Naïve Bayes classifier (by up to 9% to 15% in data sets C,F,M,0). Secondly, Dougherty et.al., found that even simple discretization schemes (e.g. EWD §2.3.3) can be competitive with more sophisticated schemes. We see the same result here where, in $\frac{13}{15}$

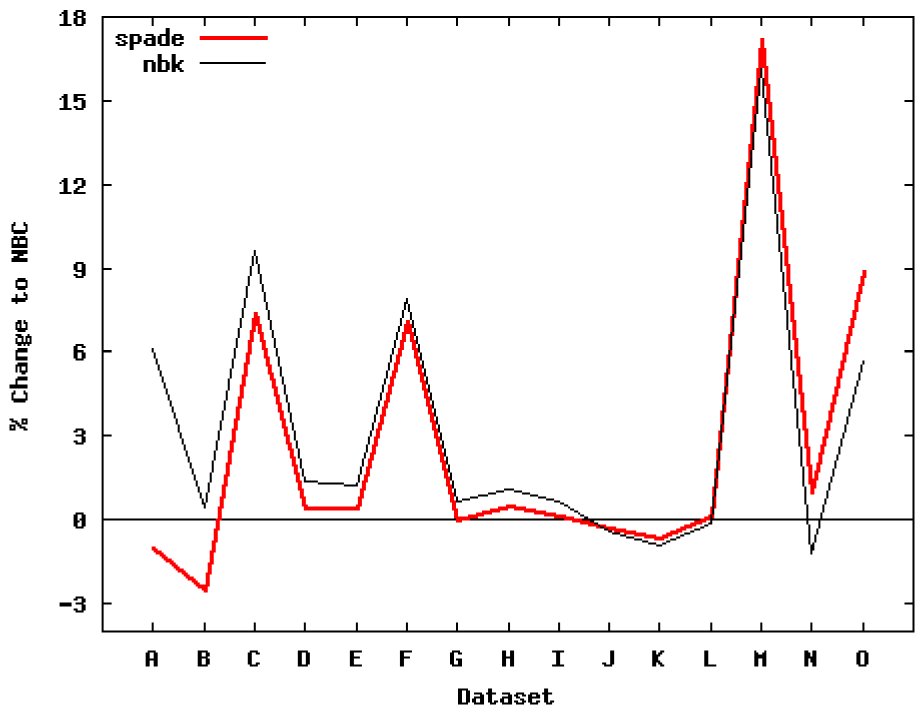


Figure 4.6: Comparing SPADE and kernel estimation. Data sets: A=vowel, B=iris, C=ionosphere, D=echo, E=horse-colic, F=anneal, G=hypothyroid, H=hepatitis, I=heart-c, J=diabetes, K=auto-mpg, L=waveform-5000, M=vehicle, N=labor, O=segment.

of these experiments, SPADE's mean improvement was within 3% of John and Langley's kernel estimation method. Thirdly, in two cases, SPADE's one scan method lost information and performed worse than assuming a single Gaussian. In data set A, the loss was minimal (-1%) and in data set B SPADE's results were still within 3% of kernel estimation. In our view, the advantages of SPADE (incremental, one scan processing) compensates for its occasional performing worse than the state-of-the-art alternatives which require far more memory.

Chapter 5

SAWTOOTH: Learning on Huge Amounts of Data

After the successful development of SPADE with its incremental capabilities, the next step to a simple but powerful scalable classifier is to integrate it to a Bayesian learner that, instead of reading a whole dataset and generate a single classification model, it reads each instance and updates the model (Bayesian table) at a time, very quickly. Such a scheme is not only useful for context where data is continuously generated and classified, but it could serve as the basis of simple unsupervised learning and other data mining techniques as we explain in §6.1.

5.1 Scaling-up Inductive Algorithms

Provost and Kolluri, in [**?**], distinguish three main methods for scaling up inductive algorithms. Firstly, there are *relational representation* methods that reject the assumption that we should learn from a single memory-resident table. Such tables can be built by joining and denormalizing other tables. Denormalization echoes the same datum in multiple locations so the joined tables are larger than the original tables. For example, joining and denormalizing three tables of data expanded the storage requirements from 100MB to 2.5GB files [?, p25]. Various relational representation learning methods exist such as hierarchical structure learners, inductive logic programming, or methods that take more advantage of the DBMS that gathers the data together for the learner.

Returning to the single-table approach, another class of scale-up methods are those based on *faster algorithms* that (e.g.) exploits *parallelism*; or that target some *restricted representations* such as a decision tree of limited depth; or that *optimize search* by, say, combining greedy search for extensions to the current theory with a fast pruning strategy.

A third class of scale-up methods perform *data partitioning* methods to learn from (e.g.) some *subset of the attributes* such as those selected via attribute relevance studies; or from *subsets of the instances* selected via, say, duplicate compaction or stratified sampling method where minority class(es) are selected with greater frequency and most of the majority class ignored.

Our technique uses the classic sequential data partitioning method called *windowing*. This particular technique is used in the FLORA1...FLORA4 systems [**?**]. The basic schema for FLORA is that newly arrived examples are pushed into the start of sliding *window* of size w while the same number of older examples are popped from the end. Adding examples can confirm or refute a generalization in the current theory. Adding *or* removing examples can trigger *relearning* from the examples in the window. As long as w is small relative to the rate of concept drift, then this procedure guarantees the maintenance of a theory relevant to the last w examples.

SAWTOOTH's windowing scheme is much simpler than FLORA. A single theory is carried across each window and SAWTOOTH takes no action when data is removed from a window. Newly arrived data is classified according to the current theory. Then they become training examples and the learner updates. If the performance of the learner has not changed after e.g. a *Stable* number of windows (we considered stable theories that remained at or above current performance three times in a row), the learning is said to have *plateaued*. Learning is then disabled and the current predictive model is frozen. Data from subsequent windows is then processed using the frozen model. Learning is reactivated if the performance seen in the current window is significantly different to the prior stability point. On reactivation, a new theory is learned from the current window onward, until stabilization is achieved, updating the frozen theory. The process repeats forever. The algorithm has its name since the resulting behavior often looks like a SAW-

TOOTH curve: performance is initially low and rises sharply during the first few windows. After a concept shift, the performance drops sharply, and then rises again to a new plateau. We called each window an *ERA* and it is a fixed user-defined parameter in SAWTOOTH's most current implementation.

Two problems with windowing systems like FLORA and SAWTOOTH are (1) the computational cost of relearning and (2) if w (ERA) is too small or too large, the learning may never find an adequate characterization of the target concept. Similarly, if w (ERA) is too large, then this will slow the learner's reaction to concept drift.

SAWTOOTH uses two mechanisms to reduce problem (1). Firstly, SAWTOOTH turns off learning while the performance of the system is stable. Secondly, SAWTOOTH uses a learner that can update its knowledge very quickly.

5.2 Concept Drift

In the real world concepts change over time. Similarly, the underlying data distribution coming from real world processes, also vary. These instabilities often make theories learned on old data inconsistent with new data, and frequent updating of the theory is required. This problem, known as *concept drift*, complicates the task of learning a model from data and requires a different approach than the one used for batch learning [**?**].

In the particular case of Bayesian classifiers, if we use them for incremental (time series) learning, the early saturation of their learned theory becomes a serious problem for adapting to changing concepts. Every time a new instance is processed, Bayesian classifiers increment counters that later are used to calculate the most likely class (see §2.2.4). This technique has an adverse effect on changing distributions since the adjustments to the new distribution may take a long time depending on how early they change. That is, the later a variation occurs, the longer it takes to adjust to new data. Moreover, if a Bayesian classifier is trained and re-trained on the same concept several times, then it takes longer to adjust to the concept if its classification changes.

As a simple example, let us suppose that NBC generated a theory that classifies a dataset with classes "0" and "1". If the theory is generated on several hundred instances from which 100 instances classified a certain combination of attribute values as class "1", then, if the distribution changes, and that same combination is now of class "0", it would take the learner another 100 instances of the same combination of attribute values to adjust and classify new instances as class "0". Such effect would make adjustments on the learned theory slowly, and in some cases even more slowly than the rate at which the distribution changes, decreasing the classification accuracy on new data.

A more realistic scenario, on the same problem, would be if the learner has been trained on thousands of instances where the majority belongs to a particular class and a small portion belongs to the remaining class (or classes). If the data distribution changes because of a small number of attribute variations, then it would take very long time to adjust to them in the same way it does in the example presented above. This is very undesirable in incremental learning, but works well on batch classification, where examples are assumed to be drawn from a single, static distribution.

In order to overcome the concept drift problem, we need to avoid training on already learned concepts so the adjustment to newer ones are easier and take less time. One study on the early stability of real world datasets helps us devise a technique, the windowing technique explained earlier, to avoid over-training the learner and over-fit the theory to initial concepts. The study is presented next.

5.2.1 Stability

During the initial phases of our investigation, we compared several of our attempts to incremental learning against C4.5 and Naïve Bayes, using the incremental 10x10-fold cross-validation technique explained in Chapter 3. From the resulting learning curves we conclude that there are three types of data sets according to the early learning saturation or stability.

First we have data sets where learning fails all together. Learners fail to produce useful theories; therefore their accuracies are very low. Examples are depicted in Figure 5.1

The second group, which is the biggest one, plateaus early, meaning that after twenty to thirty percent of the data (sometimes earlier) is seen, no

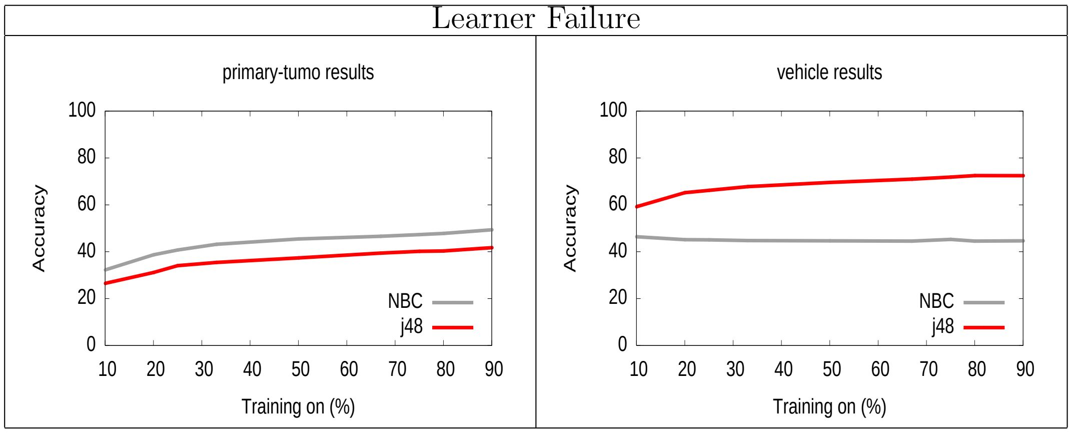


Figure 5.1: Data sets where learning accuracy is low. Learning fails.

significant performance improvement is achieved. Data sets from this group stabilize at around 300 instances, as shown in Figure 5.2.

The third group contains data sets where the more data the better the theory learned. Figure 5.3

Even though we show stability results based on NBC and C4.5, stability is present in a variety of datasets regardless of the learning algorithm. For example, Figure 5.4 summarizes stability results across two different classification concepts. On the left-hand-side it shows the performance of two discrete class classifiers, while on the right-hand-side it presents the results of two linear regression algorithms.

Figure 5.2 and Figure 5.4 suggest that, for many data sets and many learners, learning could proceed in *windows* of a few hundred instances. Learning could be disabled once performance peaks within those windows. If the learner's performance falls off the plateau (i.e. due to concept drift), it could

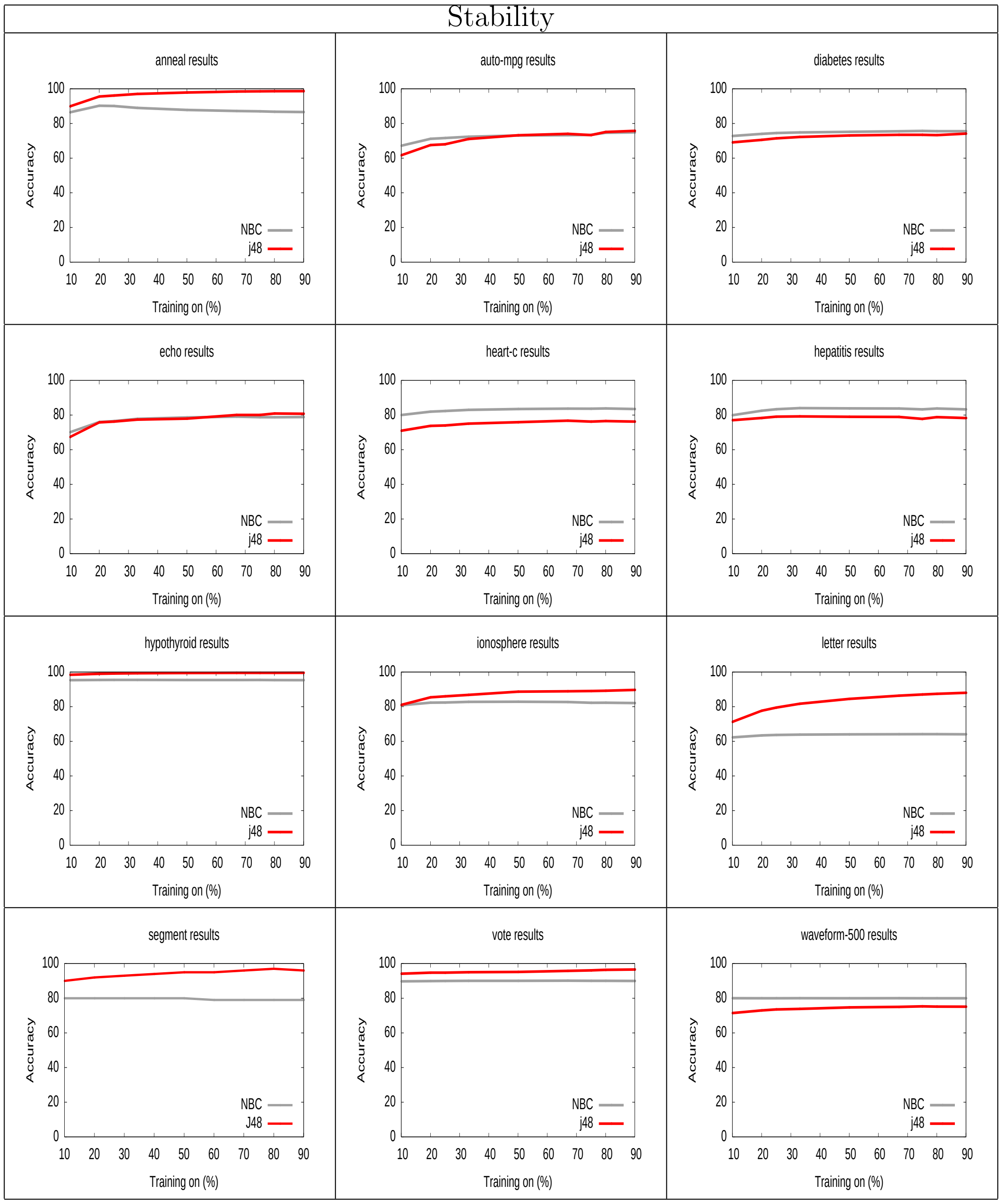


Figure 5.2: Data sets where useful theories are learned on less than 30 percent of the data. The exhibit *early plateaus*.

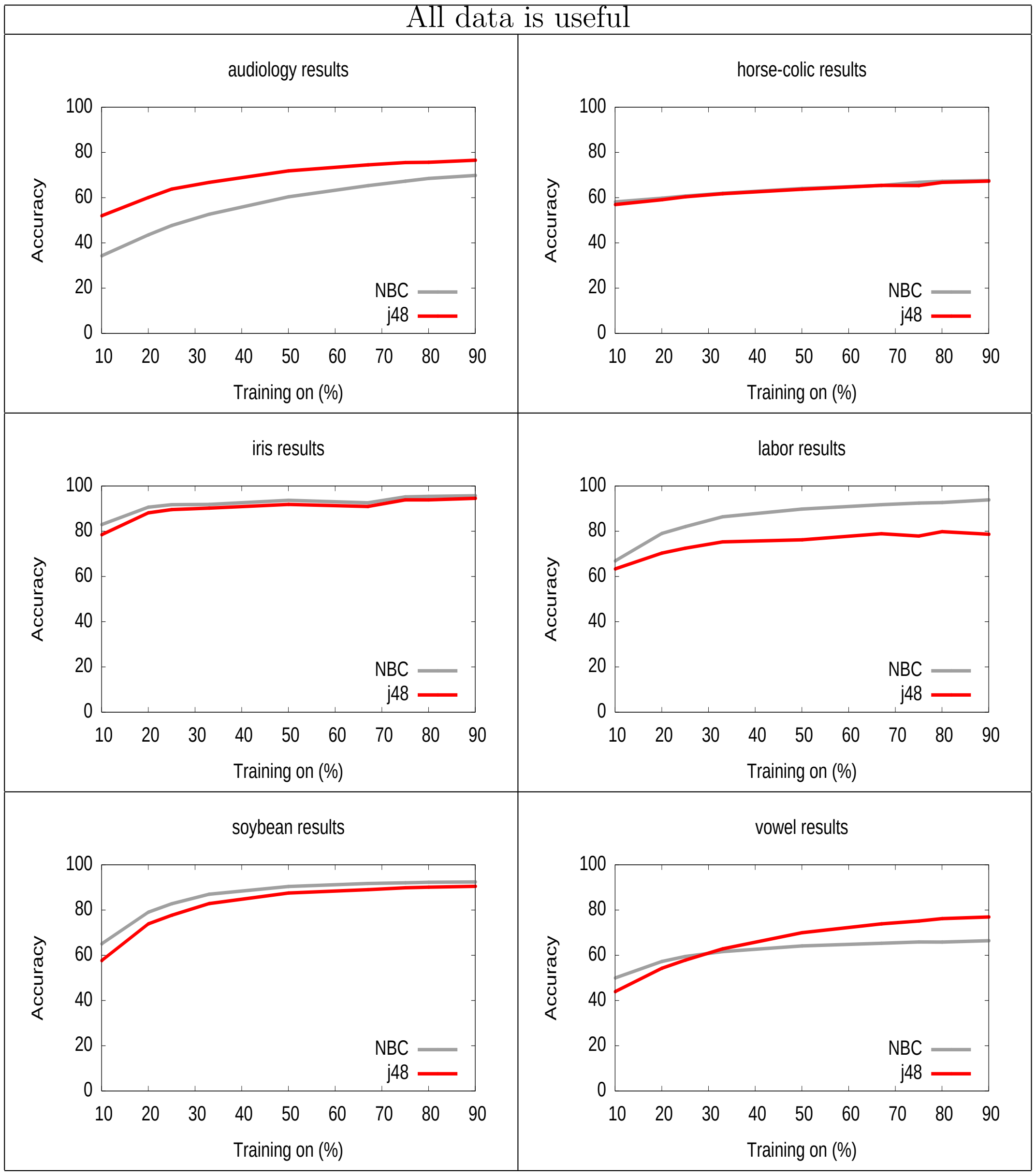


Figure 5.3: Data sets where classification accuracy increases proportional to the number of instances used for training.

update its predictive model by learning from the new instances in the current window. Given early plateaus like those in Figure 5.2 and Figure 5.4, this new learning should take only a few more hundred instances. Further, since learning only ever has to process a few hundred instances at a time, this approach should scale to very large data sets.

Figure 5.2 and Figure 5.4 are not the first reports of early plateaus in the data mining literature (though, to the best of our knowledge, it is the first report of early plateaus in M5' and LSR). Oates and Jensen found plateaus in 19 UCI data sets using five variants of C4.5 (each with a different pruning method) [**?**]. In their results, six of their runs plateaued after seeing 85 to 100% of the data. This is much later than our results, where none of our data sets needed more than 70% of the data.

One possible reason for our earlier plateaus is the method used to identify start-of-plateau. From Figure 5.2 we determined stability by eye-balling the slope of the learning curve (IF $slope \approx 0$, THEN stable), and Figure 5.4 detected plateaus using t-tests to compare performance scores seen in theories learned from M or N examples ($M < N$) and reported start-of-plateau if no significant (α=0.05) difference was detected between the N and the last M with a significant change. On the other hand, Oates and Jensen scanned the accuracies learned from $5, 10, 15\%$ etc. of the data looking for three consecutive accuracy scores that are within 1% of the score gained from a theory using all the available data. Note that regardless of *where* they found plateaus, Oates and Jensen's results endorse our general thesis that, often, learning need not process all the available examples.

We are not motivated to explore different methods for detecting start-of-plateau. The results below show that learning using our start-of-plateau detector can produce adequate classifiers that scale to very large data sets.

In summary, we may be using too much data to train our learners. In the majority of cases where useful results were obtained, there seems to be little benefit in learning from more than 30% of the data. In all cases, little improvement was seen after learning from more than 300 instances. These results imply that data mining from data sets may be quite simple. Read the data in a buffer of N instances. Train the learner on the instances of the current buffer until its theory stabilizes. Keep buffering and testing the incoming data, but do not learn from it. If the learner's performance ever drops from a stable plateau, assume that the data distribution has somehow changed and relearn from the last buffer. Repeat forever while only keeping N instances in the buffer.

That is exactly what we target with SAWTOOTH, a simple classification algorithm that oversees the performance of the current predictive model and updates it every time it fails. As a side effect, it indicates whether the current predictive model is valid for the oncoming data.

5.3 Algorithm

So far, we have seen how classification, discretization and concept drift detection works. The algorithm introduced in this section is the integration of these concepts and the final product of this thesis. The actual code is provided in §A. It has been implemented in AWK for a better comprehension and simplicity.

As we explained earlier in §2.3.2, data mining algorithms expect preformatted data, so they can process it correctly. SAWTOOTH has very low formatting requirements as it only necessitates a header line containing the attribute names separated by comma (',') and preceded by an asterisk ('*') if continuous. The class attribute should be the last one in the header and it is required to be discrete[*]. It also expects each example in the form of a line (row) with each attribute value separated by comma (',') and corresponding to the attribute named in the header at its position (column). Missing values need to be denoted as a single question mark ('?'), to be handled appropriately. Even though the attribute values are not required beforehand, we have also implemented the algorithm so it can handle Weka's ".arff" files.

The user-defined parameters this algorithm requires are:

SUBBINS: This user-defined variable affects discretization only. It specifies the number of ranges (or bins) created every time an unknown value is encountered. It prevents continuous values from being partitioned

[*]We have left the discretization of the class attribute for future work

into small number of bins. It assures a finer-grain discretization.

ERA: It is the size of the window, the buffer that holds a subset of the data for classification and testing at a single point in time. Theoretically, a bigger ERA is preferred for data whose distribution is expected to change gradually over time. Smaller ERA is better when data changes drastically over short periods of time.

The remaining parameters are initialized within the algorithm and dynamically change over the run. They are:

- **Affecting the classifier:**

  Total: Counts the total number of instances seen since the beginning of the execution.

  Classified Counts the number of classified instances within the current ERA.

  Correct: Counts the number of instances correctly classified within the current ERA.

  totalCorrect: Global counter of the total number of correctly classified instances.

  totalClassified: Global counter of the total number of classified instances.

  maxAttributes: Holds the number of attributes of the current dataset.

  classes: Array that stores the class values.

Frequencies: Matrix that holds the Bayesian table. It stores the counters for the number of seen values per attribute per class.

Seen: Secondary table that stores the number of times each attribute value is seen across all classes.

AttValues: A table that stores the number of values per attribute.

Prediction: Stores the class value returned by the classifier at every instance.

- **Affecting SPADE (discretizer):**

minInstance: Specifies the recommended minimum number of instances to be present in each partition.

maxInstance: Specifies the recommended maximum number of instances to belong to a single bin.

maxBands: Specifies the maximum number of bands (bins) allowed to exist at every point in time.

Bands: Table or matrix that holds the counters for the number of bins generated by SPADE. One counter per continuous attribute.

Numeric: Array that holds a flag for each attribute indicating whether it is continuous or discrete.

MIN: Stores the minimum values seen so far by SPADE for each continuous attribute.

MAX: Stores the minimum values seen so far by SPADE for each continuous attribute.

5.3.1 Processing Instances

SAWTOOTH starts by reading the header of the dataset and extracts some information. The header must contain the name of each attribute separated by a comma where the last attribute is the class. Continuous attributes must be distinguishable from discrete ones by a preceding asterisk. No other information is required.

During this initialization process the learner sets the values of *maxAttributes*, *Numeric*, and the starting values *Min* and *Max* for each continuous attribute. SAWTOOTH then proceeds to accept instances and collect them in a buffer of size *ERA*. Upon entering the buffer, instances are classified and counts are stored in the variables *Classified* and *Correct*. Once the buffer is full, its classification performance is statistically compared to the learner's previous performance using the *standardized test statistic* [**?**]. This test allows us to statistically compare two sets of outcomes from the same distribution with a certain *significance level* (α). In other words, it allows us to place the current accuracy estimation in a confidence interval. Let us model the situation as follows.

Standardized Test Statistic for Stability

Let S be the number of classification successes in $n = ERA$ independent trials having the binomial distribution $b(n = ERA, p = S)$. The null hypothesis, H_0, states that there is no change in the underlying distribution of the data. That means that the current ERAs mean classification accuracy μ has not dropped since the previous ERA. The alternative hypothesis, H_1, says that the underlying distribution has changed, that is, the mean classification accuracy has dropped compared to the mean classification accuracy of all processed data μ_0. [†]. The decision rule is that we reject H_0 and accept H_1 if the mean classification accuracy μ for the current ERA drops with a significance level of $\alpha = 0.05$. To achieve a test with this significance level, we choose the critical region as $\overline{X} \leq c$ such that $\alpha = 0.05 = P(\overline{X} \leq c; \mu = \mu_0)$. Since under H_0, $\overline{X}$ is $N[np, np(1-p)]$ after the binomial distribution is approximated to the normal. Then, the critical region is calculated using the formula:

$$-z(\alpha) \geq \frac{\overline{X} - \mu_0}{\sigma/\sqrt{n}} \tag{5.1}$$

where $z(0.05) = 1.645$, $\mu_0 = ERA \times S$, $\sigma = ERA \times S(1-S)$.

The formula in Equation 5.1 is called the *standardized test statistic.* There SAWTOOTH takes the difference between the sample mean and the hy-

[†]If the classification accuracy actually rises, it does not mean that the distribution of the data has changed and that we need to update the current theory by learning on the buffer, it simply means that the current theory is so similar that it is classifying better

pothesized value and "standardizes" it by the standard deviation of $\overline{X}$. If the observed value $z = (\overline{x} - \mu_0)/(\sigma/\sqrt{n})$ is a large negative value, even smaller than $-z(\alpha) = -z(0.05) = -1.645$, SAWTOOTH rejects H_0 in favor of $H_1 : \mu < \mu_0$, affirming that the distribution has changed. Otherwise, if the standardized test statistic is larger than $-z(\alpha)$, then there is not enough evidence to reject H_0, and thus the distribution is considered stable.

Armed with this test statistic, SAWTOOTH can choose whether the current buffer distribution matches what it has seen so far or if it is changing; in other words, if the learned model is stable.

SAWTOOTH considers its theory to be stable if the estimated accuracy of the current buffer lies within the confidence interval given by the standardized test statistic at the 0.05 significance level. This is the criteria by which the learner decides to either learn and forget or just forget.

Learning and Forgetting

In a previous section (§5.2) we analyzed the incremental properties of the Bayesian classifiers. From that analysis and the stability results shown in §5.2.1, we conclude that the simplest method to handle concept drift is to prevent learner over-training. This can be accomplished by only training the learner from instances where classification fails. This will also take care of the instances that it has not seen since it is not able to classify them correctly.

SAWTOOTH windowing technique activates a *cruise control* when its classification has been stable for two consecutive ERAs. While it is activated, the learner classifies the new incoming instances before they are buffered and when the buffer is full, SAWTOOTH's performance on the current buffer is compared to the overall accuracy since the last instability. If the learned model is still stable, then empty the buffer and repeat the process. Otherwise, if the classification accuracy drops, the cruise mode is deactivated and training on the current buffer is performed until stability is regained. This way, the theory is updated to the most actual state and it is ready for classification of new data. This cycle is performed forever or until the last instance is processed.

As an example, the dataset used to generate the results in Figure 5.5 was provided by Dr. Bojan Cukic. The data was generated from a flight simulator software at West Virginia University. It is composed of 4800 instances, four continuous attributes describing flight parameters, and six discrete class values: "nominal", "error1", "error2", "error3", "error4", and "error5". The "nominal" class describes a normal flight scenario. Each of the "error" classes describes a major malfunction of the aircraft. It is organized so the first 800 instances are "nominal", and then a sequence of 600 "nominal" 200 "error" instances are repeated five times, once per each different error.

While this dataset was not generated for the purpose of classification[‡], it serves us well for showing the saturation effect.

For this test we ran SAWTOOTH with $ERA = 100$, and $SUBBINS = 5$

[‡]It is actually used by Yan Liu, at the WVU Lane Department of Computer Science and Electrical Engineering, for research on a different kind of machine learning called *anomaly detection*

in both cases (left and right), but the cruise control was omitted for the results on the left. In the figure, *ERA* is the number of processed buffers. *Mode* refers to the class of the current buffer. *Inst* is the number of processed instances. *Correct* is the number of correctly classified instances in the buffer. *Clssfd* says the number of instances in the buffer. *Accuracy* tells us the proportion of correctly classified instances over the number of instances in the buffer. *State* is “Stable” if the Accuracy of the buffer is significantly equal or better than the overall accuracy, “Unstable” otherwise. Finally, *Train()* informs whether the buffer is learned or not. This last column makes the difference between the two runs of the algorithm.

These results corroborate our theory that over-training the learned theory could have negative effects. For example, when the learner classifies the first buffer of “error1” instances, in both cases it fails to recognize the class since it has never seen it. Now, when the second buffer of “error1” instances is classified, the learner that has trained on less instances of the majority class performs better than the one that has been trained on all the data. This effect is repeated when the learner gets to “error2”, “error3”, and “error5”. The only exception is found when the learner processes the “error4” instances, were in both cases SAWTOOTH fails to classify them even after processing the first buffer of instances belonging to the same error. This positive effect does not come alone though, it affects the classification of the majority class. When learning in all eras (left) the nominal instances are always classified correctly, whereas SAWTOOTH classifies such instances with inferior accuracy. This is acceptable as long as the positive effect remains greater than the negative one.

5.4 Algorithm Evaluation

5.4.1 SAWTOOTH on Changing Distributions

The first test we perform is based on the results of the flight simulator data. In this case we want to see the performance and recovery of SAWTOOTH on a dataset with concept drift. Data was taken from the simulator in ERAs of 100 instances. Each error mode lasted two ERAs, and for three times the simulator returned to each error mode. That is, we repeated three times the same dataset used in the previous section.

The top of Figure 5.6 shows the results of SAWTOOTH's stability tests as well as when SAWTOOTH enabled or disabled learning. Each error mode introduced a period of instability which, in turn, enabled a new period of learning.

The *first* time SAWTOOTH sees a new error mode (at eras 15, 23, 31, 39, and 47), the accuracy drops sharply and after each mode, accuracy returns to a high level (usually over 90%). The *second* time SAWTOOTH returns to a prior error mode (at eras 63, 71, 79, 87 and 95), the accuracies drop, but only very slightly.

Three features of Figure 5.6 are worthy of mentioning:

- The large drop in accuracy when entering new context means that SAWTOOTH can be used to recognize new contexts (watch for the large drops). In terms of certifying an adaptive system, this is a very significant result: learning systems can alert their uses when they are *leaving the region of their past competency.*

- There is no such large drop when SAWTOOTH returns to old contexts. That is, SAWTOOTH can *retain knowledge of old contexts* and reuse that knowledge *when contexts re-occur.*

- The accuracy stabilizes and SAWTOOTH mostly disables the learner between drifts. That is, for much of Figure 5.6 the SAWTOOTH "learner" is *not learning at all.*

These three points cover two of the seven standard data mining goals mentioned in the introduction of this thesis (§1.2): (D5) Can forget, and (D6) Can remember. Also, since SAWTOOTH "freezes" the predictive model when it is stable, the current best answer is readily available at any time, therefore it also supports goal (D4) On-line.

5.4.2 Batch Data

We also assess the performance of SAWTOOTH against batch learning. We have chosen twenty-one discrete-class datasets from the UCI repository [?]. These datasets are small so we can run NBC, NBK and C4.5 from the WEKA toolkit. In this comparison, we do not expect SAWTOOTH to be the best among the compared algorithms simply because it was not designed to handle such small datasets and its contribution is more towards scalability. What we do expect is a comparable to slightly better performance than NBC due to the early plateaus of some of the datasets used, but more importantly, because of SPADE.

Figure 5.7 shows a comparison between our algorithm and C4.5, NBC, NBK. From the graph, notice that SAWTOOTH clearly wins five times and losses two times against NBC. It wins in vehicle, letter, segment, ionosphere, and anneal. All but one of these datasets, ionosphere, have more than 800 instances, and all but one, anneal, have all continuous attributes (except the class). Also, they all fall in the *early plateau* group from the stability results, therefore, it is not a coincidence that SAWTOOTH does better in these datasets.

On the other hand, SAWTOOTH clearly looses in vowel and audiology. Not surprisingly, these two datasets fall in the *use all data* of Figure 5.3. On the remaining datasets, SAWTOOTH performs better than NBC and close to NBK as expected. What we did not expect was to be exactly in the middle of the average accuracy of NBC and NBK. This suggests that we can improve the overall accuracy of Naïve Bayes using the SAWTOOTH approach even in small datasets. Such results are noteworthy because SPADE is not only scalable to huge datasets, but it can also be used in smaller ones achieving results similar to the state-of-the-art batch learners. Therefore, it embraces one of the remaining four uncovered goals: SAWTOOTH is (D7) Competent.

5.4.3 Case Study: The KDD Cup 99 Classification Contest

The remarkable difference between SAWTOOTH and batch learning algorithms is encountered when they face large datasets like the KDD Cup 1999 datasets [?] on network intrusion detection. The training set is 750MB in size while the test set size is 45MB. SAWTOOTH classifies such dataset using at most 3720 KB (3.5MB) of memory, while Weka's implementation of the Naïve Bayes and the C4.5 classifiers cannot read the dataset into memory. Weka even has trouble training on the physics dataset from the KDD Cup 2004, which is 46.7MB.

SAWTOOTH's simplistic approach is very efficient and is able to handle huge amounts of data as we shall see in the KDD Cup 1999 case study presented next. We do not include the KDD Cup 2004 study because the labeled test set has not been released.

The task for the KDD Cup 1999 contest was to learn a classification theory to distinguish between different connections in a computer network for intrusion detection learning.

Two data sets were provided. One, the training set, is a collection of about five million connection records with 24 attack types (classes) and 40 attributes (from which 34 are continuous). The test set contains 311,029 instances from a different probability distribution than the training data. It has 38 classes (14 additional) and the same 40 attributes as the training set.

It is also important to note that the test instances are **not** all independent. There are 77,291 distinct test examples out of the 311,029 present in the dataset. This is important because it is a clear violation of the independence assumption of Bayesian classifiers.

Classes from both sets are divided into four main categories:

[**0**] normal: No attack.

[**1**] probe: Surveillance and other probing.

[**2**] DOS: denial-of-service.

[**3**] U2R: unauthorized access to local super-user (root) privileges.

[**4**] R2L: unauthorized access from a remote machine.

The class distributions of the datasets are:

Class	**Train**	**Test**
0	19.69%	19.48%
1	0.83%	1.34%
2	79.24%	73.90%
3	0.01%	0.07%
4	0.23%	5.20%

The classification results submitted were scored according to the following cost matrix:

	normal	probe	DOS	U2R	R2L
normal	0	1	2	2	2
probe	1	0	2	2	2
DOS	2	1	0	2	2
U2R	3	2	2	0	2
R2L	4	2	2	2	0

The 24 submitted entries were ranked according to their average classification cost as shown in Figure 5.8. This measurement is calculated by a simple script that generates a confusion matrix based on the predicted class and the actual class labels. Then this confusion matrix is dot multiplied by the cost matrix. That is, each value in the confusion matrix is multiplied by the corresponding value in the cost matrix.

Figure 5.8 is sorted in ascending order ranging from 0.2356 to 0.9414. "The first significant difference between entries with adjacent ranks is between the 17th and 18th best entries. The difference is very large: $0.2952 - 0.2684 = 0.0268$, which is about nine standard errors. One could conclude that the best 17 entries all performed well, while the worst 7 entries were definitely inferior."[§]

Figure 5.9 shows the confusion matrix obtained by the winning group. Their approach involved an ensemble of 50x10 C5 decision trees using a cost-sensitive bagged boosting technique[¶]. It took several runs of the commercially available C5 (Supposedly better than C4.5) algorithm, implemented in C, on many sub-samples of the training set. Although they did not use the whole training set, and the algorithms used were implemented efficiently, their final run took over a day on a dual-processor 2x300MH Ultra-Sparc2 machine, with 512MB of RAM, running Solaris 5.6.

[§]Extracted from the KDD Cup 99 results page available at http://www.cs.ucsd.edu/users/elkan/clresults.html

[¶]More details are given at http://www.ai.univie.ac.at/~bernhard/kddcup99.html

5.4.4 SAWTOOTH Results

Motivated by the large scale of the datasets offered for this task, we ran SAWTOOTH and obtained the confusion matrix and average classification cost in Figure 5.10.

SAWTOOTH's average classification cost then falls between places 15 and 16 of the submitted entries. That means that SAWTOOTH is within the group that performed well according to the judges. SAWTOOTH's average classification cost was close to the winning entrant score; very similar to entrants 10, 11, 12, 13, 14, 15, 16; and better than entrants 18,19,20,21,22,23,24. These results are encouraging since SAWTOOTH is a much simpler tool than the winning entry. Even though SAWTOOTH is written in interpreted scripting languages (gawk/bash), it processed all 5,300,000 instances in one scan using less than 3.5 Megabytes of memory. This took 11.5 hours on a 2GHz Pentium 4, with 500MB of RAM, running Windows/Cygwin. We conjecture that this runtime could be greatly reduced by porting our toolkit to "C".

Another encouraging result is the *# bins with tally=X* plot of Figure 5.11. One concern with SPADE is that several of its internal parameters are linked to the number of processed instances; e.g. *MaxBins* is the square root of the number of instances. The 5,300,000 instances of KDD'99 could therefore, in the worst case, generate over 2000 bins for each continuous attribute. This worst-case scenario would occur if each consecutive group of SUBBINS number of continuous values would have different values from the previously seen groups and are sorted in ascending or descending order. If this unlikely combination of events does not occur then the resulting bins would have tallies than MININST, encouraging it to merge with the next bin. In all of our experiments, we have never seen this worst-case behavior. In KDD'99, for example, SPADE only ever generated 2 bins for 20 of the 34 continuous attributes. Also, for only one of the attributes, did SPADE generate more than 50 bins. Lastly, SPADE never generated more than 100 bins.

From this case study we can mark off two of the remaining three goals left to achieve by SAWTOOTH. It is clear from this test that it is: (D2) Small, since it requires about the same amount of memory to process any of the smaller UCI datasets as it took this huge one; and (D3) One scan, requiring a single pass over the set of instances to draw conclusions. There is only one standard data mining goal that has not been met: (D1) Fast. At this point, we cannot assure that SAWTOOTH runs in linear time. This is because SPADE's bin generation method is linked to the size of the dataset, as explained in Chapter 4. What we did see is that, experimentally, SPADE does not generate anywhere near the worst case (square root of the number of instances), and the number of bins is usually small.

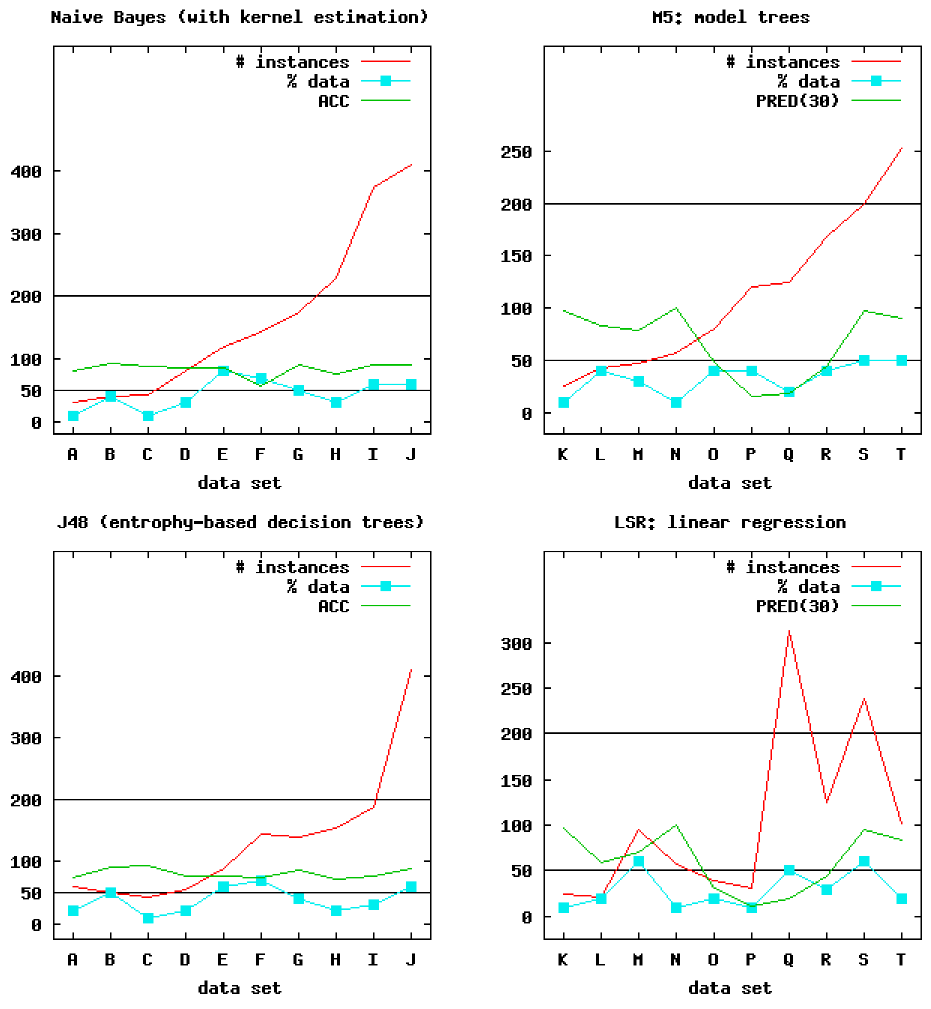


Figure 5.4: *R=10*N=10* incremental cross validation experiments on 20 UCI data sets [?]. A:heart-c, B:zoo; C:vote; D:heart-statlog; E:lymph, F:autos. G:ionosphere, H:diabetes, I:balance-scale, J:soybean, K:bodyfat. L:cloud, M:fishcatch, N:sensory, O:pwLinear, Q:strike, R:pbc, S:auto-mpg, T:housing. Data sets A..J have discrete classes and are scored via the *accuracy* of the learned theory; i.e % successful classifications. Data sets K..T have continuous classes and are scored by the *PRED(30)* of the learned theory; i.e. what % of the estimated values are within 30% of the actual value. Data sets are sorted according to how many instances were required to reach plateau using nbk and C4.5 (left-hand side) or M5' and LSR (right-hand side).

```
ERA     Mode  Inst  Correct  Clssfd  StblAccy  State     train()
  1  nominal   100        0     100  0.000000  initial   yes
  2  nominal   200      100     100  1.000000  stable    yes
  3  nominal   300      100     100  1.000000  stable    yes
  4  nominal   400      100     100  1.000000  stable    yes
  5  nominal   500      100     100  1.000000  stable    yes
  6  nominal   600      100     100  1.000000  stable    yes
  7  nominal   700      100     100  1.000000  stable    yes
  8  nominal   800      100     100  1.000000  stable    yes
  9  nominal   900      100     100  1.000000  stable    yes
 10  nominal  1000      100     100  1.000000  stable    yes
 11  nominal  1100      100     100  1.000000  stable    yes
 12  nominal  1200      100     100  1.000000  stable    yes
 13  nominal  1300      100     100  1.000000  stable    yes
 14  nominal  1400      100     100  1.000000  stable    yes
 15   error1  1500        0     100  0.928571  unstable  yes
 16   error1  1600       60     100  0.300000  stable    yes
 17  nominal  1700      100     100  0.533333  stable    yes
 18  nominal  1800      100     100  0.650000  stable    yes
 19  nominal  1900      100     100  0.720000  stable    yes
 20  nominal  2000      100     100  0.766667  stable    yes
 21  nominal  2100      100     100  0.800000  stable    yes
 22  nominal  2200      100     100  0.825000  stable    yes
 23   error2  2300        0     100  0.733333  unstable  yes
 24   error2  2400       69     100  0.345000  stable    yes
 25  nominal  2500      100     100  0.563333  stable    yes
 26  nominal  2600      100     100  0.672500  stable    yes
 27  nominal  2700      100     100  0.738000  stable    yes
 28  nominal  2800      100     100  0.781667  stable    yes
 29  nominal  2900      100     100  0.812857  stable    yes
 30  nominal  3000      100     100  0.836250  stable    yes
 31   error3  3100        0     100  0.743333  unstable  yes
 32   error3  3200       35     100  0.175000  stable    yes
 33  nominal  3300      100     100  0.450000  stable    yes
 34  nominal  3400      100     100  0.587500  stable    yes
 35  nominal  3500      100     100  0.670000  stable    yes
 36  nominal  3600      100     100  0.725000  stable    yes
 37  nominal  3700      100     100  0.764286  stable    yes
 38  nominal  3800      100     100  0.793750  stable    yes
 39   error4  3900        0     100  0.705556  unstable  yes
 40   error4  4000        3     100  0.015000  stable    yes
 41  nominal  4100      100     100  0.343333  stable    yes
 42  nominal  4200      100     100  0.507500  stable    yes
 43  nominal  4300      100     100  0.606000  stable    yes
 44  nominal  4400      100     100  0.671667  stable    yes
 45  nominal  4500      100     100  0.718571  stable    yes
 46  nominal  4600      100     100  0.753750  stable    yes
 47   error5  4700        0     100  0.670000  unstable  yes
 48   error5  4800        4     100  0.020000  stable    yes
82.3617
```

```
ERA     Mode  Inst  Correct  Clssfd  StblAccy  State     train()
  1  nominal   100        0     100  0.000000  initial   yes
  2  nominal   200      100     100  1.000000  stable    yes
  3  nominal   300      100     100  1.000000  stable    yes
  4  nominal   400      100     100  1.000000  stable    no
  5  nominal   500      100     100  1.000000  stable    no
  6  nominal   600      100     100  1.000000  stable    no
  7  nominal   700      100     100  1.000000  stable    no
  8  nominal   800      100     100  1.000000  stable    no
  9  nominal   900      100     100  1.000000  stable    no
 10  nominal  1000      100     100  1.000000  stable    no
 11  nominal  1100      100     100  1.000000  stable    no
 12  nominal  1200      100     100  1.000000  stable    no
 13  nominal  1300      100     100  1.000000  stable    no
 14  nominal  1400      100     100  1.000000  stable    no
 15   error1  1500        0     100  0.928571  unstable  yes
 16   error1  1600       80     100  0.400000  stable    yes
 17  nominal  1700      100     100  0.600000  stable    yes
 18  nominal  1800      100     100  0.700000  stable    yes
 19  nominal  1900      100     100  0.760000  stable    no
 20  nominal  2000      100     100  0.800000  stable    no
 21  nominal  2100      100     100  0.828571  stable    no
 22  nominal  2200      100     100  0.850000  stable    no
 23   error2  2300        0     100  0.755556  unstable  yes
 24   error2  2400       99     100  0.495000  stable    yes
 25  nominal  2500      100     100  0.663333  stable    yes
 26  nominal  2600      100     100  0.747500  stable    yes
 27  nominal  2700      100     100  0.798000  stable    no
 28  nominal  2800      100     100  0.831667  stable    no
 29  nominal  2900      100     100  0.855714  stable    no
 30  nominal  3000      100     100  0.873750  stable    no
 31   error3  3100        0     100  0.776667  unstable  yes
 32   error3  3200       81     100  0.405000  stable    yes
 33  nominal  3300      100     100  0.603333  stable    yes
 34  nominal  3400      100     100  0.702500  stable    yes
 35  nominal  3500       92     100  0.746000  stable    no
 36  nominal  3600       85     100  0.763333  stable    no
 37  nominal  3700       88     100  0.780000  stable    no
 38  nominal  3800       94     100  0.800000  stable    no
 39   error4  3900        0     100  0.711111  unstable  yes
 40   error4  4000        3     100  0.015000  stable    yes
 41  nominal  4100      100     100  0.343333  stable    yes
 42  nominal  4200      100     100  0.507500  stable    yes
 43  nominal  4300      100     100  0.606000  stable    no
 44  nominal  4400       96     100  0.665000  stable    no
 45  nominal  4500       97     100  0.708571  stable    no
 46  nominal  4600       91     100  0.733750  stable    no
 47   error5  4700        0     100  0.652222  unstable  yes
 48   error5  4800       33     100  0.165000  stable    yes
83.8085
```

Figure 5.5: Difference between learning using all data (left) and learning on the unstable portions of it(right). See how the performance on new classes degrades as more training is provided.

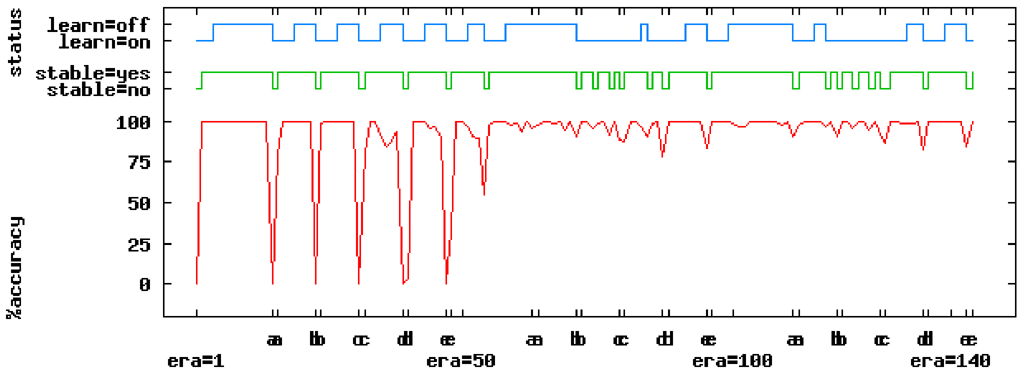
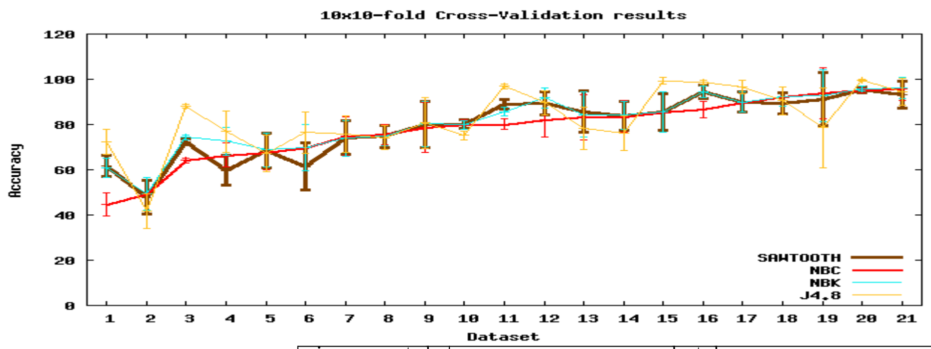
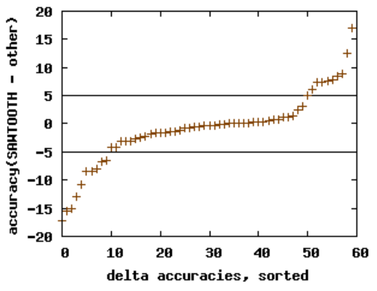


Figure 5.6: SAWTOOTH and Concept Drift



			incremental	batch		
#	data set	instances	SAWTOOTH	NBC	NBK	J4.8
1	vehicle	846	61.62 ± 4.73	44.64 ± 4.95 -	60.77 ± 4.28	72.42 ± 5.52 +
2	primary-tumor	339	47.83 ± 7.34	49.37 ± 7.09	49.37 ± 7.09	41.73 ± 7.63 -
3	letter	20000	72.44 ± 1.26	64.05 ± 1.11 -	74.28 ± 1.08 +	88.01 ± 0.83 +
4	vowel	990	59.72 ± 6.51	66.42 ± 5.31 +	72.56 ± 6.33 +	76.91 ± 9.14 +
5	horse-colic	368	68.41 ± 7.55	67.58 ± 7.64	68.81 ± 7.44	67.31 ± 8.18
6	audiology	226	61.42 ± 10.45	69.84 ± 10.35+	69.84 ± 10.35 +	76.55 ± 9.10 +
7	auto-mpg	398	74.24 ± 7.48	74.85 ± 8.37	73.95 ± 7.83	75.73 ± 8.17
8	diabetes	798	74.70 ± 4.81	75.54 ± 4.36	75.10 ± 4.64	74.15 ± 5.41
9	echo	130	80.08 ± 10.09	78.85 ± 11.01	80.23 ± 9.98	80.77 ± 11.34
10	waveform-5000	5000	80.16 ± 1.77	80.00 ± 1.93	79.84 ± 1.95	75.12 ± 2.03 -
11	segment	2310	88.91 ± 2.24	80.04 ± 2.34 -	85.73 ± 2.09 -	96.87 ± 1.19 +
12	ionosphere	351	89.54 ± 5.09	82.09 ± 7.70 -	91.73 ± 4.31 +	89.66 ± 5.28
13	hepatitis	155	85.67 ± 9.15	83.28 ± 10.04 -	84.38 ± 9.92	78.28 ± 9.49 -
14	heart-c	303	83.74 ± 6.46	83.50 ± 6.97	84.16 ± 6.24	76.23 ± 7.71 -
15	kr-vs-kp	3196	85.49 ± 8.12	85.66 ± 9.01	85.66 ± 9.01	99.18 ± 1.49 +
16	anneal	898	94.50 ± 2.97	86.61 ± 3.59 -	94.49 ± 2.47	98.69 ± 1.02 +
17	vote	435	89.98 ± 4.32	89.96 ± 4.35	89.96 ± 4.35	96.58 ± 2.97 +
18	soybean	683	89.38 ± 4.76	92.42 ± 4.21 +	92.42 ± 4.21 +	90.49 ± 6.04
19	labor	57	91.13 ± 11.76	93.87 ± 11.26	92.67 ± 11.46	78.67 ± 17.65 -
20	hypothyroid	3772	95.36 ± 1.30	95.30 ± 1.06	95.93 ± 0.98	99.53 ± 0.32 +
21	iris	150	93.13 ± 6.03	95.67 ± 5.14 +	96.13 ± 4.85 +	94.53 ± 5.87
	MEAN		79.40	78.07	80.86	82.26



learner	win - loss	win	loss	ties
J48	13	28	15	20
nbk	7	17	10	32
SAWTOOTH	-7	12	19	32
NB	-13	9	22	32

Figure 5.7: *mean* ± *standard deviations* seen in 10*10-way cross validation experiments on UCI Irvine data sets running NBC NBK and J4.8. The graph shows the performance on the learners on the 21 datasets. The plot bottom-right sorts the differences in the accuracies found by SAWTOOTH and all the other learners. Some of those differences are not statistically significant: the "+" or "-" in the table denote mean differences that are significantly different to SAWTOOTH at the $\alpha = 0.05$ level. The significant differences between all the learners are shown in the win-loss statistics of the bottom-right table.

1	0.2331	7	0.2474	13	0.2552	19	0.3344
2	0.2356	8	0.2479	14	0.2575	20	0.3767
3	0.2367	9	0.2523	15	0.2588	21	0.3854
4	0.2411	10	0.2530	16	0.2644	22	0.3899
5	0.2414	11	0.2531	17	0.2684	23	0.5053
6	0.2443	12	0.2545	18	0.2952	24	0.9414

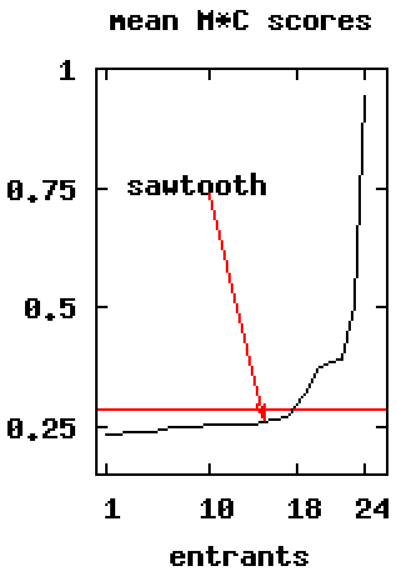
Figure 5.8: Average classification cost results from the 24 submitted entries to the KDD Cup 1999.

```
     Predicted      0       1       2       3       4     %correct
Actual     \-------------------------------------------------------
 0         |   60262    243      78       4       6       99.5%
 1         |     511   3471     184       0       0       83.3%
 2         |    5299   1328  223226       0       0       97.1%
 3         |     168     20       0      30      10       13.2%
 4         |   14527    294       0       8    1360        8.4%
           |
%correct   |    74.6%  64.8%  99.9%  71.4%  98.8%

Average classification cost: 0.2331
Correctly Classified: 288349
Accuracy: 92.708%
```

Figure 5.9: Detailed classification summary attained by the winning entry.

```
      predicted   0        1        2        3       4       %correct
actual  \---------------------------------------------------------------
   0     |    60163      324       52        2      52        99.3%
   1     |      739     3257      170        0       0        78.2%
   2     |     6574      880   222399        0       0        96.8%
   3     |      136       82        0       10       0         4.4%
   4     |    16115       73        0        0       1         0.0%
         |
 %correct|      71.9%    70.6%    99.9%    83.3%   1.9%

Average classification cost: 0.25985
Correctly Classified: 285830
Accuracy: 91.898%
```

Figure 5.10: SAWTOOTH detailed classification summary on the KDD Cup 99 dataset.



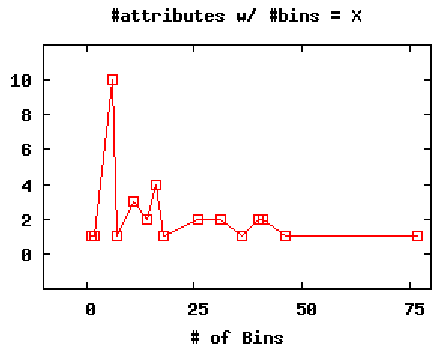


Figure 5.11: SAWTOOTH and the KDD'99 data

Chapter 6

Conclusion and Future Work

In this thesis, we have found that mining on huge datasets can be successfully accomplished in a very simplistic manner. Our original objective of building a simple classifier able to handle huge or infinite data sets with possible numeric attributes while validating the theory learned and possibly adapting to changes in the underlying distribution of the data, is accomplished with SAWTOOTH. Our results show that its performance is comparable to Naïve Bayes on batch data, but with very low memory requirements. This is remarkable in the sense that it can handle virtually unlimited amounts of data while preserving the Bayesian advantages of incremental updatability, readiness of the learned model, and robustness to missing values. The addition of an on-line discretizer, not only allows it to handle continuous values in the data, but it can actually increase the accuracy of the learner, and eliminates the assumption of normality within each attribute's distribution.

Holte argues for the *simplicity-first* approach to data mining; i.e. researchers should try simpler methods before complicating existing algorithms [?]. While Provost and Kolluri endorse "simplicity-first", they note in their review of methods for scaling up inductive algorithms that, "it is not clear now much *leverage* can be obtained through the use of simpler classifiers to guide subsequent search to address *specific deficiencies* in their performance" [?, p32].

This thesis has been a simplicity-first approach to scaling up data miners. We have *levered* two features of Naïve Bayes classifiers that make them good candidates for handling large datasets: fast updates of the current theory and small memory foot print. Several *deficiencies* with Naïve Bayes classifiers have been addressed: incremental discretization and dynamic windowing means that Bayes classifiers need not hold all the data in RAM at one time.

Our algorithm works via one scan of the data and can scale to millions of instances. It is much simpler than other scale-up methods such as FLORA or the winner of KDD'99. Even so, it performs as well as many other data mining schemes (see Figure 5.11). Also, when used on smaller data sets, it performs within ±5% of other commonly used schemes (see Figure 5.7). Further, the same algorithm without any modifications can be used to detect concept drift, to repair a theory after concept drift, and can reuse old knowledge when old contexts re-occur (see Figure 5.6).

A drawback with our learner is that we cannot guarantee that it operates in small constant time per incoming instance. Several of SPADE's internal parameters are functions of the total number of instances. In the worst case, this could lead to the runaway generation of bins. On a more optimistic note, we note that this worst case behavior has yet to be observed in our experiments: usually, the number of generated bins is quite small (see Figure 5.11).

From the results of this thesis, we can say that SAWTOOTH fully accomplishes all but one (D1 FAST) of the standard data mining goals presented in §1.2. It is a **small**, **one-scan**, **on-line**, **can forget**, **can recall**, and **competent** classification algorithm. We have also shown that the goal D1 fast is attainable experimentally.

Why can such a simple algorithm like SAWTOOTH be so competent for both small and large data sets? Our answer is that many data sets (such as all those processed in our experiments) exhibit early plateaus and such early plateaus can be exploited to build very simple learners. If a particular data set does not contain early plateaus then our simple learner should be exchanged for a more sophisticated scheme. Also, SAWTOOTH is inappropriate if concept drift is occurring *faster* than the time required to collect enough instances to find the plateau.

Finally, we recommend that other data mining researchers check for early plateaus in their data sets. If such plateaus are a widespread phenomena, then *very simple tools* (like SAWTOOTH) should be adequate for the purposes of scaling up induction.

6.1 Future Work

This research does not stop here. There are countless opportunities to fine-tune SAWTOOTH for even better performance, but more importantly, it could be expanded to help with scalability and simplicity on many different areas of machine learning.

SAWTOOTH can be improved by implementing it in a programming language, as opposed to a scripting one. It could also be improved by exploiting parallelism. One thread could update the theory while another classifies new examples on the most recently frozen model. Even a third thread could be used to handle classification of unlabeled data. Both approaches would speed-up the algorithm several orders of magnitude.

A different approach would be taking numeric class datasets and incrementally discretize the class with SPADE as an approximation to linear regression. It would be really interesting to see if what is true for discrete class classification holds in the context of numeric classes. If such algorithm's performance is comparable to the state-of-the-art linear regression algorithms, then it would simplify very complicated real world tasks present in a variety of domains ranging from stock market fluctuations, to spatial calculations.

One area of great interest is unsupervised learning. Very simple approaches to anomaly detection, where the likelihood of the novel class is tracked for sudden changes, could be developed based on the research provided by this thesis. Likelihood variation above a certain threshold could determine an anomaly on the distribution. One-class classification techniques like that is essential for the latest advancements in unmanned exploration, where mission critical components need to detect behaviors outside the nominal ones to take an action. This is the context in which the flight simulator data was generated. It was targeted to this kind of learning.

On another note, our research could also help speed-up treatment learning by changing the confidence1 measure to likelihood and making use of SPADE for discretization. According to Geletko and Menzies [?], treatment learning can be really useful for testing real world models. The only drawback is that it takes a really long time to run, and it grows exponentially with the number of instances. Treatment learning could become incremental and scalable based on our research, leading to an invaluable tool for critical validation and verification (V&V) of models.

Finally, any machine learning approach based on probabilities could take advantage of SPADE by simply embedding it and attaching it to the probability table.

Appendix A

SAWTOOTH Implementation in AWK

A.1 Interface

A.1.1 SAWTOOTH What is This?

```
copyleft() {
cat<<-EOF
sawtooth: An incremental online naive bayes classifier with
on-line, one-pass discretization, and stabilization tuning.
Copyright (C) 2004 Andres Orrego and Tim Menzies

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation, version 2.
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.
You should have received a copy of the GNU General Public License
along with this program; if not, write to the
Free Software Foundation, Inc.
59 Temple Place - Suite 330
Boston, MA  02111-1307, USA.

EOF
}
```

A.1.2 Usage

```
usage() {
      cat<<-EOF
Usage: sawtooth [FLAG]... FILE
SawTooth

Flags:
  -h        print this help text
  -l        copyright notice
  -X  N     run one of N demos. N=0 means run all demos
  -v        verbose mode (shows bayes table & run times)
  -p        Brief Mode (Shows predicted and actual classes)
  -t <file>   training file specifier
  -T <file>   test file specifier
  -s <number> Number of sub divisions of a discretized bin (default 5)
  -e <number> Number of instances that determines an ERA (cache size)
EOF
}
```

A.1.3 Motivation

```
#During our research we have performed classification experiments
#on many datasets and we have noticed three accuracy (3) patterns
#in respect to the number of instances:

#We may be using too much data to train our learners.
#In the majority of  cases where useful results were obtained,
# there seemed little benefit is learning from more than
# 30% of the data.
# In all cases, little improvement was seen after learning from
# 300 instances. See www.scant.org/bill/sequence.html

#What are the implications of this study? For one thing,
#data mining from data sets may be quite simple. Learn a theory
#until it stablizes. Classify and incrementally train on I<N>
#instances until the classification accuracy stabilizes,
#then stop training but keep classifying and
#discretizing (if needed) until the mean accuracy drops
#or the end of the dataset is reached
```

A.1.4 Installation

Updates

```
#To update the package, run the install procedure, described below,
#again.

#After the first install
#there will exist a directory sawtooth/local directory.
#Subequent updates may overwrite all other directories EXCEPT
#for sawtooth/local. So do your own work in sawtooth/local.
```

Install

```
#Requires a UNIX system (LINUX most preferred and Solaris, least)
#with a bash shell interpreter (I have gotten it before using ksh
#but bash is best.)

#Also
#requires a variable $SCANT
#pointing to a directory for all the  scant packages. Then define
#certain useful search paths. This can be set many
#ways including adding these lines to $HOME/.bashrc

# SCANT="$HOME/someSubDirectory"
# PATH="$SCANT/share/local:$SCANT/share/bin:$PATH"
# AWKPATH="$SCANT/share/local:$SCANT/share/bin:$AWKPATH"
# export SCANT PATH AWKPATH

#Download www.scant.org/sawtooth/var/sawtooth.zip sawtooth.zip
#into a temp directory. Then:

# unzip -d $SCANT sawtooth.zip #creates $SCANT/sawtooth, $SCANT/share
# $SCANT/share/bin/publish

#Check the pathnames in $SCANT/share/etc/sharerc. If they look wrong,
#add fixed pathways to I$SCANT/share/local/sharerc. Next, try
#getting the help text:

# cd $SCANT/sawtooth/bin
# ./sawtooth -h
```

```
#If that works, try running a demo:

# cd $SCANT/sawtooth/bin
# ./sawtooth -X

#If that works, you should see YYY:
```

A.1.5 Source code

SAWTOOTH Configuration

```
. $SCANT/share/bin/config sawtooth
```

Main driver

```
sawtoothMain() {
  $awk -f sawtooth.awk \
       ERA=$era \
       Verbose=$verbose \
       Brief=$brief \
       SUBBINS=$subbins \
       $train \
       $test
}
```

Command Line Processing

```
defaults(){
    demo=""
    era=150
    verbose=0
    brief=0
    subbins=5
    train=""
    test=""
}

defaults

while getopts "hlvt:T:ps:e:" flag
do case "$flag" in
      l)  copyleft; exit;;
      h)  usage; exit ;;
      v)  verbose=1;;
      t)  train="Pass=1 $OPTARG";;
      T)  test="Pass=2 $OPTARG";;
      p)  brief=1;;
      s)  subbins="$OPTARG";;
      e)  era="$OPTARG";;
   esac
done
shift $(($OPTIND - 1))

if [ -z "$train" ]
then
  echo "training file not specified"
  usage
  exit
else
  sawtoothMain
fi
```

A.2 The Worker

Initialization

```
BEGIN {
#Command line arguments:
# Verbose  # for verbose mode
# Brief    # for "brief mode"
# SUBBINS  # Number of subdivisions per new value seen (discretization)
# ERA      # Cache size (expresed in number of instances)

  FS=" *, *";  # Field separator is a comma and surrounding blank-spaces

#Internal globals:
  Total=0;     # count of all instances
  Classified=0;# count of currently classified instances
  Correct=0;   # count of currently correctly classified instances
  totClsfd=0;  # count of all classified instances
  totCorrect=0;# count of all correctly classified instances
  stblClsfd=0; # count of correctly classified insts. since stable
  stblCrrct=0; # count of classified insts. since stable
  guess="";    # Stores the current class prediction
  MaxAtt=0;    # count of number of attributes
  StTime = 0;  # Start time
  TrTime = 0;  # Training Time
  TsTime = 0;  # Test Time
  TtTime = 0;  # Total Time
  MININST=0;   # minimum number of instances per band split
  MAXINST=0;   # max number of instances
  MINBANDS=0;  # minimum number of bands per attribute
  MAXBANDS=0;  # max number of bands
  trained=0;   # Flag that determines whether first ERA was learned.
  stDevAcc=0.0;# Standar deviation of the accuracies
  meanAcc=0.0; # Mean of the accuracies
  zTest=0.0;   # ZTest results
# MaxBands     # table of counters for attribute ranges
# NumAtt       # table of index for numeric attributes
# Classes      # table of class names/frequencies
# Freq         # table of counters for values in attributes in classes
# Cache        # Temporary Bayesian table.
# Seen         # table of counters for values in attributes
# Attributes   # table of number of values per attribute
  StTime = systime(); # Start timer.
} # END Initialization
```

Training

```
Pass==1 {
  if(FNR==1){                       # Processing header of train file.
    init();      # Checks for file type and attribute characteristics.
    next;
  }
  do{
    if ($0~/^%/ || NF != MaxAtt) continue;
    guess = classify();
    if ($NF == guess) {Correct++;}       # Classification and testing.
    Classified++;

    for(a=1; a<=NF; a++){                # Caching instances
      Cache[Classified,a] = $a;
    }

    if (Classified>=ERA){break;}
    if (getline <= 0) {break;}
  } while(Pass==1)

  if(!trained){
    trained = 1;
  }
  else{
    stblClsfd += Classified; # Update counters
    stblCrrct += Correct;
    totClsfd += Classified;
    totCorrect += Correct;
    stDevAcc = StDev(Classified,Correct); # calculate test parameters.
    newMeanAcc = Correct; #n*p=classified*(correct/classified)=correct
    oldMeanAcc = stblCrrct*Classified/stblClsfd; #mu_0=n*p_0

    # Standardized test statistics Z(0.01) = 2.326
    zTest=ZTest(newMeanAcc,oldMeanAcc,stDevAcc,Classified);
    if (zTest < -2.326){    # Not stable
      Stable = -1;
      stblCrrct=Correct;    # Reset counters of Stable
      stblClsfd=Classified;
    }
    else{                   # Stable
      Stable++;
    }
  }

  if(Stable < 3){            # IF stability is preserved
    train(Cache,Classified); # THEN train on the cache.
  }
  delete Cache;              # Resete Cache

  Correct = Classified = 0;  # Reset counters of ERA

  if(NumDS){       # IF dataset has numeric attributes
    updateBands(); # THEN Update discretization table after each ERA.
  }
} # END Training
```

Testing

```
Pass==2 {                                        # Classification time!
  if(FNR==1){                        # Processing header of test file.
    TrTime = systime();
    test();   # Checks for matching file type and attributes in init().
    if (NumDS){
      updateBands()
    }
    if (Verbose){
print "\n\nDISCRETE VALUES AFTER TRAINING";
      printDVals();
    }
    totCorrect=0; totClsfd=0;   # Reset counters of Totals when testing
    next;
  }
  if ($0~/^%/ || NF!=MaxAtt) next;

  guess = classify();  # Classification

  if ($NF == guess) totCorrect++;  # Classification and testing.
  totClsfd++;

  if (Brief)
  print guess;
} # END Testing
```

Post-Processing

```
END {
  TsTime = systime();
  if (Verbose) {                              # When in verbose mode.
    print "\n\nDISCRETE VALUES AFTER TESTING";
    printDVals();                # Prints the discretized attributes.
    printTable();                        # Prints the bayesian table.
    print "Number of Classes:",Attributes[MaxAtt];
    print "Correctly Classified Instances : " totCorrect;
    print "Total Number of Instances Classified : " totClsfd;
    printf "accuracy: "
   }
  if(!Brief) print totCorrect*100/totClsfd;
  TtTime = systime();
  if (Timer){              # When in timing mode, times are displayed.
    printf "\nTraining Time: %-2.2dm %-2.2ds\n",
      (TrTime-StTime)/60,(TrTime-StTime)%60;
    printf "Testing Time : %-2.2dm %-2.2ds\n",
      (TsTime-TrTime)/60,(TsTime-TrTime)%60;
    printf "Total Time   : %-2.2dm %-2.2ds\n",
      (TtTime-StTime)/60,(TtTime-StTime)%60;
  }
}
```

A.2.1 User Defined functions

Standardized Test Statistic

```
function ZTest(newMean,oldMean,stDev,totInst){
  if(stDev == 0) return newMean - oldMean;
  return (newMean-oldMean)/(stDev/sqrt(totInst));
}
```

Standar Deviation

```
function StDev(n,corr,       p){
  if(n > 1){
    p=corr/n;
    return sqrt(n*p*(1-p));
  }
  return 0;
}
```

Data File Initialization

```
function init(    i){
  if (FILENAME~/.mbc$/){  # .MBC files.
    for(i=1;i<=NF;i++){
      header[i]=$i;       # Stores the attribute value name i.
      if ($i ~ /^\*/) {
        NumAtt[i]=1;      # Determines that the field i is numeric.
        NumDS=1;
        Min[i,1]=1e+32;   # Minimum number in the field i.
        Max[i,1]=-1e+32;  # Maximum number in the field i.
      }
      else NumAtt[i]=0;   # field i is discrete.
    }
    MaxAtt=NF;            # Total number of attributes.
  }
  if (FILENAME~/.arff$/){ # .ARFF files (WEKA).
    FS = " ";
    i=1;
    while($0!~/^ *@r/ && $0!~/^ *@R/) getline;
    while($0!~/^ *@d/ && $0!~/ *^@D/){
      if ($0~/^ *@a/ || $0~/^ *@A/) {
        header[i] = $2;
        if ($3!~/^\{/){
          NumAtt[i]=1;
          NumDS=1;
          Min[i,1]=1e+32;
          Max[i,1]=-1e+32;
        }
  i++;
      }
      getline;
    }
    MaxAtt = i-1;
    FS=" *, *";
  }
}
```

Validating the Test Datafile

```
function test(    i){
  if (FILENAME~/.mbc$/){
    for(i=1;i<=NF;i++){
      if(header[i]!=$i){  # Matches atts names with train file.
  print "Error. Invalid testing file. Exiting...";
  exit 1;
      }
    }
  }
  if (FILENAME~/.arff$/){
    while($0!~/^ *@d/ && $0!~/^ *@D/){ # Skips file header.
      getline;
    }
  }
}
```

Bayesian Training Function

```
function train(array,size,      a,val,i,c) {
  for(a=1;a<=size;a++){
    Total++;
    c=array[a,MaxAtt];
    Classes[c]++;
    for(i=1;i<=MaxAtt;i++) {
      if (array[a,i] ~ /?/) continue;
      if (NumAtt[i]){
  val=discretize(i,array[a,i]+0);
      }
      else val=array[a,i];
      Freq[c,i,val]++
if (++Seen[i,val]==1) Attributes[i]++;
    }
  }
}
```

Bayesian Classification Function

```
function classify(       i,temp,what,like,c,prior,m,inst) {
  m=2;
  like = 0;
  for(i=1;i<NF;i++) {
    if (NumAtt[i] && $i !~ /?/)
```

```
      inst[i]=discretize(i,$i+0); # Discretization step
    else inst[i]=$i;
  }
  for(c in Classes) {
    prior=(Classes[c]+1)/(Total+Attributes[MaxAtt]);
    temp=1.0;
    for(i=1;i<NF;i++) {
      if ( inst[i] ~ /?/ )
        continue;             # ignores unknown values
      val=inst[i];
      temp *= ((Freq[c,i,val]+(m*prior))/(Classes[c]+m));
    }
    temp *= prior;
    if ( temp >= like ) {like = temp; what=c}
  }
  return what;
}
```

SPADE's Bin Creation Function

```
function discretize(fld,item,    i,j,k,subdiv){

  # numeric value lies between min max seen so far.
  if (item>=Min[fld,1] && item<=Max[fld,1]) {
    # search for the band who contains it and return its position.
    return find(fld,item);
  }

  # creating first Band. It is of the form (item,item]
  if (item<Min[fld,1] && item>Max[fld,1]){
    i=0;               # The index for the first band is 0
    Band[fld,i,1]=item; # Band is the array where "fld" is the attribute,
    Band[fld,i,2]=item; # " i" is the position of the band, "1" is the
                       # lower limit and "2" is the upper limit.
    Min[fld,1]=item; # Min is an array to store the overall Min value
    Min[fld,2]=i;    # and its position, "1," and "2" respectively.
    Max[fld,1]=item;   # Max is analogous to Min.
    Max[fld,2]=i;
    MaxBands[fld]++;  # The number of bands is now 1
    return i;
  }

  # If the numeric value is less than the min seen so far.
  if (item<Min[fld,1]){
    subdiv=((Band[fld,Min[fld,2],2] - item) / SUBBINS); #calc subdiv
    Band[fld,Min[fld,2],1]=item+((SUBBINS-1)*subdiv); # Update lwrlimit
    i=Min[fld,2]-SUBBINS;
    Band[fld,i,1]=item;
    Band[fld,i,2]=item;
    Min[fld,1]=item;
    Min[fld,2]=i;
    MaxBands[fld]++;
    i++;
    for(j=1; j<SUBBINS; j++){
      Band[fld,i,1]=item+((j-1)*subdiv);
      Band[fld,i,2]=item+(j*subdiv);
      i++;
      MaxBands[fld]++;
    }
    return Min[fld,2];
  }

  # If the numeric value is greater than the max seen so far.
  if (item>Max[fld,1]){
    subdiv = ((item - Band[fld,Max[fld,2],2]) / SUBBINS);
    i=Max[fld,2]+1;
    for(j=1; j<SUBBINS; j++){
      Band[fld,i,1] = Max[fld,1]+((j-1)*subdiv);
      Band[fld,i,2] = Max[fld,1]+(j*subdiv);
      MaxBands[fld]++;
      i++;
    }
    Band[fld,i,1]=Max[fld,1]+((SUBBINS-1)*subdiv);
    Band[fld,i,2]=item;
    MaxBands[fld]++;
    Max[fld,1]=item;
    Max[fld,2]=i;
    return i;
  }
}
```

Updating Bins of a Single Numeric Attribute

```
function updateBand(i,        j,k,l,m,n,p,old,busy,temp){
  l=0;m=0;
  n=Max[i,2];
  old = Min[i,2]+1;
  for(j=old+1; j<=n; j++){
    if(Seen[i,j]<(MAXINST-Seen[i,old])&& Seen[i,old]<MININST){
      Band[i,j,1]=Band[i,old,1];
      Band[i,old,1]=""; #lower limit
      Band[i,old,2]=""; #upper limit
      Seen[i,j]+=Seen[i,old];
      Seen[i,old]=0;
      for(k in Classes){
Freq[k,i,j] += Freq[k,i,old];
Freq[k,i,old]=0;
      }
    }
    else{
      busy[++l,"min"] = Band[i,old,1];
      busy[l,"max"] = Band[i,old,2];
      busy[l,"seen"] = Seen[i,old];
      for(k in Classes)
busy[l,k] = Freq[k,i,old];
    }
    old = j;
  }
  m=Min[i,2]+1;
  p=0;
  for(j=m; j<m+l; j++){
    Band[i,j,1] = busy[++p,"min"];
    Band[i,j,2] = busy[p,"max"];
    Seen[i,j] = busy[p,"seen"];
    for(k in Classes)
      Freq[k,i,j] = busy[p,k];
  }
  Band[i,j,1]=Band[i,n,1];
  Band[i,j,2]=Band[i,n,2];
  Seen[i,j]=Seen[i,n];
  for(k in Classes)
    Freq[k,i,j] = Freq[k,i,n];
  Max[i,2] = j;
  MaxBands[i] = (j-m)+2;
  for(j++ ;j<=n; j++){
    delete Band[i,j,1];
    delete Band[i,j,2];
    delete Seen[i,j];
    for(k in Classes)
      delete Freq[k,i,j];
  }
}
```

Updating the Bins of All Numeric Attributes

```
function updateBands(           j){
  MININST = sqrt(Total);      # Sets variables needed for updating
  MAXINST = MININST * 2;      # the values of the current attribute
  MAXBANDS =  MININST; #* SUBBINS;
  for(j=1; j<=MaxAtt; j++){
    if(NumAtt[j]){
      if (MaxBands[j]>MAXBANDS){
updateBand(j);
      }
    }
  }
}
```

Displaying the Predictive Model

```
function printTable(    i,j,k){
  printf "\n| %17s B A Y E S I A N    T A B L E %17s|", " ", " ";
  printf "\n| %-15.15s | %-15.15s | %-9.9s | %-6.6s | %-6.6s |\n",
         "ATTRIBUTE","VALUE","CLASS","FREQ","SEEN";
  for(i=1;i<=MaxAtt;i++){
    print Attributes[i];
    for(j in Freq){
      split(j,k,SUBSEP);
      if (k[2]==i && Seen[k[2],k[3]] > 0){
        if (NumAtt[i])
          printf "| %-15.15s | (%2.4f,%-2.4f] | %-9.9s | %6d | %6d |\n",
                 header[i], Band[i,k[3],1],Band[i,k[3],2], k[1],
                 Freq[k[1],k[2],k[3]], Seen[k[2],k[3]];
        else
          printf "| %-15.15s | %-15.15s | %-9.9s | %6d | %6d |\n",
                 header[i], k[3], k[1], Freq[k[1],k[2],k[3]],
                 Seen[k[2],k[3]];
      }
    }
  }
}
```

Displaying SPADE's Bins

```
function printDVal(j,    i,k){
  print "Att Name:", header[j] ,
        "\tNumber of Bands:",MaxBands[j],
        "\tMin Val:",Min[j,1],
        "\t\tMax Val:",Max[j,1];
  for(i=Min[j,2]; i<=Max[j,2]; i++){
        printf "( %s , %s ]=%d ",Band[j,i,1],Band[j,i,2],Seen[j,i];
   }
   print "\n";
 }
```

Displaying All Attribute Conversions

```
function printDVals(       i){
  for(i=1; i<=MaxAtt; i++)
    if (NumAtt[i]) printDVal(i);
}
```

Finding a Number's Bin

```
function find(fld,item,         left,mid,right){
  if (item == Min[fld,1]) {
    return Min[fld,2];
  }
  left = Min[fld,2];
  right = Max[fld,2];
  while (left < right){
    mid = int((left+right)/2);
    if (item > Band[fld,mid,2]) {left=mid+1}
    else {
      if (item == Band[fld,mid,2] || item > Band[fld,mid,1]) {
return mid;
      }
      right=mid-1;
    }
  }
  return left;
}
```

A.3 Configuration File

```
#Loads the config files in the right order.
#Lets 'local' config files override defaults in 'etc' config files

 . $SCANT/share/etc/sharerc
 [ -f  $SCANT/share/local/sharerc ] && . $SCANT/share/local/sharerc
 . $SCANT/$1/etc/${1}rc
 [ -f  $SCANT/$1/local/${1}rc ]  && . $SCANT/$1/local/${1}rc
```

```
# Defines search paths to let stuff in 'local' override default code.
# Useful for adding 'local' patches.

 PATH="$SCANT/$1/local:$SCANT/$1/bin:$PATH"
 AWKPATH="$SCANT/$1/local:$SCANT/$1/bin:$AWKPATH"

 export PATH AWKPATH
```

Bibliography

[1] S. D. Bay. Multivariate discretization of continuous variables for set mining. In *Proceedings of the 6th International Conference on Knowledge Discovery and Data Mining*, pages 315–319, 2000.

[2] P. Berka and I. Bruha. Discretization and grouping: Preprocessing steps for data mining. In *Proceedings of the Second European Symposium on Principles of Data Mining and Knowledge Discovery*, pages 239–245. Springer-Verlag, 1998.

[3] C. Blake and C. Merz. UCI repository of machine learning databases, 1998.

[4] R. Bouckaert. Choosing between two learning algorithms based on calibrated tests. In *Machine Learning, Proceedings of the Twentieth International Conference*, 2003.

[5] L. Breiman. Bias, variance, and arcing classifiers, 1996.

[6] J. Cattlet. On changing continuous attributes into ordered discrete attributes. In *Proceedings of the european working session on learning*, pages 164–178, 1991.

[7] P. Domingos and M. J. Pazzani. Beyond independence: Conditions for the optimality of the simple bayesian classifier. In *International Conference on Machine Learning*, pages 105–112, 1996.

[8] P. Domingos and M. J. Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29(2-3):103–130, 1997.

[9] J. Dougherty, R. Kohavi, and M. Sahami. Supervised and unsupervised discretization of continuous features. In *Machine Learning, Proceedings of the Twelfth International Conference*, pages 194–202, 1995.

[10] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis.* John Wiley Sons, New York, 1973.

[11] L. V. Fausett. *Fundamentals of Neural Networks.* Prentice-Hall, 1 edition, 1994.

[12] U. M. Fayyad and I. H. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1022–1027, 1993.

[13] U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. Knowledge discovery and data mining: Towards a unifying framework. In *Proceedings of the Second International Conference on Knowledge Discovery and data Mining*, pages 82–88, 1996.

[14] J. H. Friedman. On bias, variance, 0/1-loss, and the curse-of-dimensionality. *Data Mininig Knowledge Discovery*, 1(1):55–77, 1997.

[15] D. Geletko and T. J. Menzies. Model-based software testing via treatment learning. In *28th Annual NASA Goddard Software Engineering Workshop (SEW'03)*, December 2003.

[16] M. A. Hall and G. Holmes. Benchmarking attribute selection techniques for discrete class data mining. *IEEE Transactions on Knowledge and Data Engineering*, 13(3), 2001.

[17] J. A. Hartigan and M. A. Wong. Statistical algorithms: Algorithm AS 136: A K-means clustering algorithm. 28(1):100–108, Mar. 1979.

[18] S. Haykin. *Neural Networks: A Comprehensive Foundation.* Prentice-Hall, 2 edition, 1998.

[19] G. E. Hinton and T. J. Sejnowsky. Learning and re-learning in boltzmann machines. In *Parallel distributed processing*, volume 1, chapter 7. 1986.

[20] R. V. Hogg and J. Ledolter. *Applied Statistics for Engineers and Physical Scientists.* Macmillan Publishing Company, 2 edition, 1992.

[21] R. C. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11(1):63 – 90, 1993.

[22] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. In *Proceedings of the National Academy of Sciences*, pages 2554–2558, 1982.

[23] C.-N. Hsu, H.-J. Huang, and T.-T. Wong. Why discretization works for naïve bayesian classifiers. In *Proc. 17th International Conf. on Machine Learning*, pages 399–406. Morgan Kaufmann, San Francisco, CA, 2000.

[24] Y. Hu. Treatment learning: Implementation and application. Master's thesis, University of British Columbia, 2003.

[25] G. H. John and P. Langley. Estimating continuous distributions in Bayesian classifiers. In *Uncertainty in Artificial Intelligence, Proceedings of the Eleventh Conference*, pages 338–345, 1995.

[26] R. Kerber. Discretization of numeric attributes. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 123–128. MIT Press, 1992.

[27] R. Kohavi. Bottom-up induction of oblivious, read-once decision graphs: strengths and limitations. In *Twelfth National Conference on Artificial Intelligence*, pages 613–618, 1994.

[28] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI*, pages 1137–1145, 1995.

[29] R. Kohavi. Scaling up the accuracy of Naive-Bayes classifiers: a decision-tree hybrid. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, pages 202–207, 1996.

[30] R. Kohavi and D. H. Wolpert. Bias plus variance decomposition for zero-one loss functions. In L. Saitta, editor, *Machine Learning: Proceedings of the Thirteenth International Conference*, pages 275–283. Morgan Kaufmann, 1996.

[31] E. B. Kong and T. G. Dietterich. Error-correcting output coding corrects bias and variance. In *International Conference on Machine Learning*, pages 313–321, 1995.

[32] W. Kwedlo and M. Kretowski. An evolutionary algorithm using multivariate discretization for decision rule induction. In *Principles of Data Mining and Knowledge Discovery*, pages 392–397, 1999.

[33] P. Langley, W. Iba, and K. Thompson. An analysis of bayesian classifiers. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 223–228. AAAI Press and MIT Press, 1992.

[34] T. Menzies, E. Chiang, M. Feather, Y. Hu, and J. Kiper. Condensing uncertainty via incremental treatment learning. In T. M. Khoshgoftaar, editor, *Software Engineering with Computational Intelligence*. Kluwer, 2003. Available from `http://menzies.us/pdf/02itar2.pdf`.

[35] T. Menzies and Y. Hu. Reusing models for requirements engineering. In *First International Workshop on Model-based Requirements Engineering*, November 2001).

[36] T. Menzies and Y. Hu. Just enough learning (of association rules): The tar2 treatment learner. In *Journal of Data and Knowledge Engineering (submitted)*, 2002. Available from `http://menzies.us/pdf/02tar2.pdf`.

[37] T. Menzies and Y. Hu. Data mining for very busy people. In *IEEE Computer*, November 2003. Available from `http://menzies.us/pdf/03tar2.pdf`.

[38] T. Menzies, J. Kiper, and M. Feather. Improved software engineering decision support through automatic argument reduction tools. In *SEDECS: The 2nd International Workshop on Software Engineering Decision Support (part of SEKE2003)*, June 2003. Available from `http://menzies.us/pdf/03star1.pdf`.

[39] T. Menzies, D. Raffo, S. on Setamanit, Y. Hu, and S. Tootoonian. Model-based tests of truisms. In *Proceedings of IEEE ASE 2002*, 2002. Available from `http://menzies.us/pdf/02truisms.pdf`.

[40] J. Mingers. An empirical comparison of selection measures for decision tree induction. *Machine Learning,3*, 1989.

[41] T. Oates and D. Jensen. The effects of training set size on decision tree complexity. In *Proc. 14th International Conference on Machine Learning*, pages 254–262. Morgan Kaufmann, 1997.

[42] F. J. Provost and V. Kolluri. A survey of methods for scaling up inductive algorithms. *Data Mining and Knowledge Discovery*,

3(2):131–169, 1999. Available from http://citeseer.ist.psu.edu/provost99survey.html.

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

[44] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1992.

[45] F. Rosenblatt. *Principles of Neurodynamics: Perceptrons and the theory of brain mechanisms*. Spartan Books, 1962.

[46] S. Salzberg. On comparing classifiers: Pitfals to avoid and a recommended approach. *Data Mining and Knowledge Discovery*, 1(3):317–327, 1997.

[47] I. W. Sandberg, J. T. Lo, C. L. Fancourt, J. C. Principe, S. Katagiri, and S. Haykin. *Nonlinear Dynamical Systems: Feedforward Neural Network Perspectives*. Willey US, 2001.

[48] J. Shavlik, R. Mooney, and G. Towell. Symbolic and neural learning algorithms: An experimental comparison. *Machine Learning*, 6:111–143, 1991.

[49] F. E. H. Tay and L. Shen. A modified chi2 algorithm for discretization. *Knowledge and Data Engineering*, 14(3):666–670, 2002.

[50] A. Tsymbal. The problem of concept drift: definitions and related work. Technical report, Trinity College Dublin, 2004. Available at http://www.cs.tcd.ie/publications/tech-reports/reports.04/TCD-CS-2004-15.pdf.

[51] G. Webb and M. Pazzani. Adjusted probability naive bayesian induction. In *Proceedings of the Eleventh Australian Joint Conference on Artificial Intelligence*, pages 285–295, 1998.

[52] G. I. Webb. MultiBoosting: A technique for combining Boosting and Wagging. *Machine Learning*, 40(2):159–196, 2000.

[53] G. Widmer and M. Kubat. Learning in the presence of concept drift and hidden contexts. *Machine Learning*, 23(1):69–101, 1996. Available from http://citeseer.ist.psu.edu/widmer96learning.html.

[54] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan-Kaufman, 1st edition, 1999.

[55] D. Wolpert. On the connection between in-sample testing and generalization error. *Complex Systems*, 6:47–94, 1992.

[56] D. H. Wolpert. Stacked generalization. Technical Report LA-UR-90-3460, Los Alamos, NM, 1990. Available from http://citeseer.ist.psu.edu/wolpert92stacked.html.

[57] Y. Yang. Discretization for naive-bayes learning. Master's thesis, Monash University, 2003.

[58] Y. Yang and G. I. Webb. Proportional k-interval discretization for naive-bayes classifiers. In *12th European Conference on Machine Learning (ECML01)*, pages 564–575, Berlin, 2001. Springer-Verlag.

[59] Y. Yang and G. I. Webb. A comparative study of discretization methods for naive-bayes classifiers. In *Proceedings of PKAW 2002: The 2002 Pacific Rim Knowledge Acquisition Workshop*, pages 159–173, Tokyo, 2002.

[60] Y. Yang and G. I. Webb. Non-disjoint discretization for naive-bayes classifiers. In *Proceedings of the Nineteenth International Conference on Machine Learning (ICML '02)*, pages 666– 673, San Francisco, 2002. Morgan Kaufmann.

[61] Y. Yang and G. I. Webb. On why discretization works for naive-bayes classifiers. In *Proceedings of the 16th Australian Conference on AI (AI 03)Lecture Notes AI 2903*, pages 440–452, Berlin, 2003. Springer.

[62] Y. Yang and G. I. Webb. Weighted proportional k-interval discretization for naive-bayes classifiers. In *Proceedings of the Seventh Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD03), Springer*, pages 501–512, Berlin, 2003. Springer-Verlag.

[63] Z.-H. Zhou. Three perspectives of data mining. *Artificial Intelligence, 2003, 143(1): 139-146*, 143(1):139–146, 2003.