Instituto de Ciências Matemáticas e de Computação

ISSN - 0103-2569

Reviewing Some Machine Learning Concepts and Methods

José Augusto Baranauskas
Maria Carolina Monard/ILTC

Nº 102

RELATÓRIOS TÉCNICOS DO ICMC

São Carlos
Fevereiro/2000

Reviewing Some Machine Learning Concepts and Methods*

José Augusto Baranauskas
Maria Carolina Monard/ILTC

University of São Paulo
Institute of Mathematics and Computer Sciences
Department of Computer Science and Statistics
Laboratory of Computational Intelligence
P.O. Box 668, 13560-970 - São Carlos, SP, Brazil
e-mail: {jaugusto, mcmonard}@icmc.sc.usp.br

Abstract

Machine Learning — ML — is a field in Artificial Intelligence where an inductive concept is learnt — a classifier — from given concept instances. One common difficulty faced by beginners in ML refers to basic terms and concepts that are assumed to be known. This work tries to minimize this difficulty, compiling into one document many terms, concepts and methods that are generally spread out across several ML bibliographies.

This work provides some definitions of *learning*, why to use ML as well as some categorization of learning systems. Definitions accomplished by examples, formulas and figures are given explaining some common terms used in ML. Since most learning systems use some kind of search, we also discuss some uninformed and informed search strategies. Resampling techniques are also described involving holdout, random, cross–validation, leave–one–out and bootstrap samples.

Another point considered is the bias variance decomposition of classifiers which may explain, in certain way, the success of ensembles like bagging, boosting and arcing, beyond others, also considered in this report.

Since in many applications it is also important that the concept description induced by a ML algorithm can be interpreted by humans, descriptions that allow for human comprehensibility expressed in the form of decision trees and decision rules are treated in some detail in this work.

Keywords: Supervised Learning, Resampling, nsembles.

February 2000

*Work partially supported by National Research Councils — CAPES and FINEP, Brazil.

This document was produced with the LaTeX typeset system and the BibTeX reference management system with help of the BibView tool (Prati et al., 1999). As with all reviewing work, it almost certainly contains errors and has plenty of room for improvements. Please report any error, typos, inconsistences, omissions and suggestions for improvements to jaugusto@icmc.sc.usp.br.

This document and possible updates can be found at the ICMC site:

ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/Rt_102.ps.zip

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

> Imagination is more important than knowledge. Knowledge is limited while imagination embraces the entire world.
>
> —*Albert Einstein, On Science*

In this work we concentrate in inductive concept learning where a concept description, or classifier, is induced from given concept instances (data). In other words, a concept description represents a generalization of given facts. We briefly introduce this sort of learning and explain the most important terms that are used in Machine Learning — ML — as well as some description languages used to describe instances, hypotheses and background knowledge. Although this work concentrates in classification tasks, most concepts also apply to regression or non-symbolic problems as well.

Being an inductive process, the correctness of the concept description cannot be guaranteed. Therefore, the concept description found should always have to be tested on new data. Several resampling techniques — including holdout, random, cross–validation, leave–one–out and bootstrap — that allow to estimate the true error of a predictor as well as a simple statistical test to measure the significance of any difference between two classifiers are also described. Since most learning systems use some kind of search, we also discuss some uninformed and informed search strategies.

Furthermore, due to the fact that the induced descriptions can be incorrect, in many applications it is also important that they can be interpreted by humans. Descriptions expressed in the form of decision trees and decision rules, treated in some detail in this work, allow for human comprehensibility.

This work is organized as follows. Section 2 gives some background on Machine Learning including several introductory concepts, embracing some definitions of learning systems, common terms, formulas and methods broadly used in this field, such as search strategies, resampling estimators, combining predictors (ensembles) and a simple test for comparing two algorithms using errors. Section 3 briefly describes induction of decision trees and Section 4 discuss the rule induction process. Finally, concluding remarks are given in Section 5.

2 Supervised Machine Learning

> The best way to learn about something is doing it.
>
> —*Maria Carolina Monard*

This section describes the supervised classification learning problem and the terms used throughout this work. Readers not familiar with Machine Learning concepts might wish to consult text books on the subject, such as (Weiss and Kulikowski, 1991; Langley, 1996; Mitchell, 1998).

2.1 Why Machine Learning?

Humans are natural observers. We usually observe one process occurring in nature, or human created, and we figure out how to understand it. Sometimes, we decide to measure some features of such a process, hoping they could help us on comprehending it.

After gathering those features they are stored into five basic storage stages:

1. human mind recording;
2. manual recording in cards or paper media;
3. capturing data into computer flat files, including text files and spreadsheets, whose primary purpose is storing data in digital form;
4. Data Base Management System — DBMS — whose primary purpose is reliable storage and fast data access;
5. data marting or data warehousing, whose main purpose is data storage for decision support.

Note that one could start in the first stage ending up in the last one, depending on the number of features and instances being stored as well as the available technology.

Once stored in a digital form, the data is generally used in a limited way, usually querying through some well-known application or report generating facility. While this traditional mode of interaction is satisfactory for well-defined queries, it is not designed to support answers to questions like:

- Is it possible to predict the behaviour of such process?
- How the data can be used to build classifiers of the underlying process which generated it?
- How to understand better the data and use it to gain some sort of advantage or make the process better?

These questions arrive naturally when the user does not know how to describe the goal in terms of a traditional query. In this case, a more natural way of interacting with the data is to state the Query By Example — QBE — typically more feasible because humans find natural to interact at level of examples (or instances). Usually this is performed by labeling a training set of instances of one class versus other classes and letting a ML system to build a classifier for distinguishing classes.

After that, one can then apply the extracted classifier to search all data (not only the training set) for situations of interest. As stated in (Breiman et al., 1984), data-derived classifications can serve for two basic purposes:

1. to predict the class label corresponding to new measurements of features as accurately as possible;
2. to understand the structural relationship between the class label and the measured features.

The next section introduces some concepts about the learning hierarchy, providing a distinction among learning systems.

2.2 The Learning Hierarchy

> It is only the limitations of the human mind that make the possible, impossible.
>
> —*Marc Drake*

Simon (Simon, 1983) defined learning as:

> "Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same task or tasks drawn from the same population more efficiently and more effectively the next time."

Weiss (Weiss and Kulikowski, 1991) defined learning as:

> "A learning system is a computer program that makes decisions based on the accumulated experience contained in successfully solved cases."

Inductive learning is accomplished by reasoning from externally supplied instances. In supervised learning, also known as pattern recognition or discrimination, the inducer is supplied with a set of training instances for which the true class labels are known. Each instance is described typically by a vector of feature values and a class label. The task of the induction algorithm is to learn a classifier that correctly classifies new unlabeled instances *i.e.*, without class label. For discrete class labels the problem is known as *classification* and for continuous class labels as *regression*. On the other hand, in unsupervised learning the classes are inferred from the data. Figure 1 shows the learning hierarchy just described where shaded nodes lead to supervised classification learning, the topic of this work.

Figure 2 shows an example of a classification process. As can be seen, background knowledge about the domain may be used when choosing the data or providing some information already known as input to the ML inducer. After induced, the classifier is usually evaluated and the classification process may be repeated if necessary. One important issue to consider is that classifiers should provide a more compact description of the concept embedded on the underlying data. In other words, if we have n instances then it is expected that the induced classifier should provide a description length less than n (or the data would describe itself better than that classifier!). For example, a decision tree induced from n instances should have less than n leaf nodes.

Another important requirement is that ML should be able to deal with imperfect data. In general, instances contain a certain amount of errors in their description or some features may have missing values.

Learning systems usually are classified into two major categories (Kubat et al., 1998a):

- *black-box* systems which develop their own concept representation *i.e.*, their internal representation cannot be easily interpreted by humans and provides neither insight nor explanation of the recognition process;
- *knowledge-oriented* systems which aim at creating symbolic structures that are human comprehensible (Michalski, 1983).

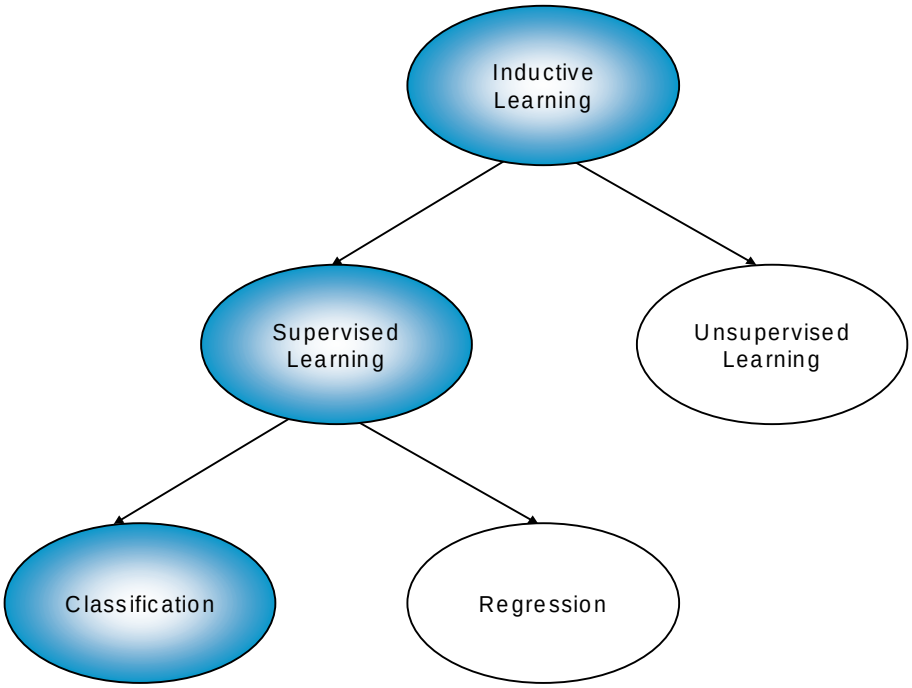
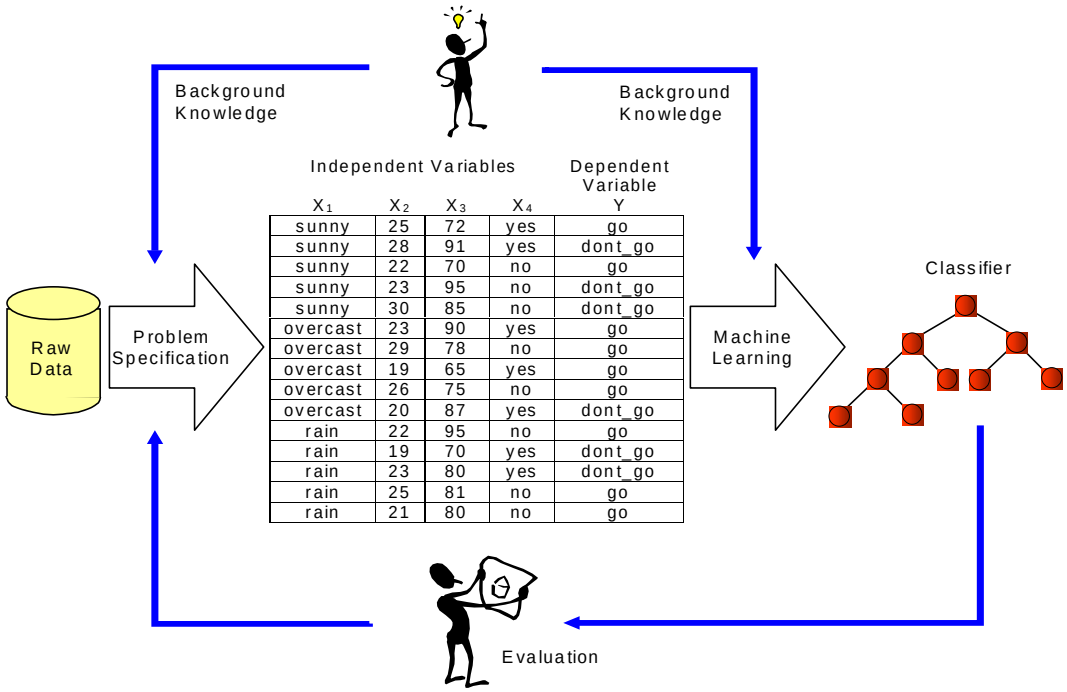


Figure 1: The learning hierarchy: shaded nodes lead to supervised classification learning, the topic of this work



X_1	X_2	X_3	X_4	Y
sunny	25	72	yes	go
sunny	28	91	yes	dont_go
sunny	22	70	no	go
sunny	23	95	no	dont_go
sunny	30	85	no	dont_go
overcast	23	90	yes	go
overcast	29	78	no	go
overcast	19	65	yes	go
overcast	26	75	no	go
overcast	20	87	yes	dont_go
rain	22	95	no	go
rain	19	70	yes	dont_go
rain	23	80	yes	dont_go
rain	25	81	no	go
rain	21	80	no	go

Figure 2: The classifier toward the right provides a compact interpretation of the data

A distinction between these two categories has been formulated in (Michie, 1988) in terms of three criteria:

- *weak criterion*: the system uses data to generate an updated basis for improved performance on subsequent data. Neural Networks and Statistical methods satisfy this criterion;
- *strong criterion*: the *weak* criterion is satisfied. Besides that, the system is able to communicate its internal representation in explicit symbolic form;
- *ultra-strong criterion*: *weak* and *strong* criteria are satisfied. Moreover, the system is able to communicate its internal representation in explicit symbolic form that can be used by a human without being aided by computer *i.e.*, using only the human mind.

In this work we shall concentrate in learning by symbolic supervised classification. The term *symbolic* indicates that classifiers should be human readable and comprehensible. The term *supervised* suggests that some process, sometimes called the teacher, has previously classified the instances in the training set. Finally, as explained before, the term *classification* denotes the fact that the class label is discrete *i.e.*, it consists of a few unordered values.

Next, some terms broadly used through this work are defined.

2.3 Definitions

All things good to know are difficult to learn.

Greek Proverb

Inducer Informally, the task of an inducer (or a learning program) is to generate a *good* classifier from a set of classified instances. The classifier can then be used to classify unlabeled instances with the goal of correctly predicting the label of each unlabeled instance. Then, the classifier can be evaluated for accuracy (Salzberg, 1995; Dietterich, 1997b), comprehensibility (Freitas, 1998b; Horst, 1999), learning speed, storage requirements, compactness and any other desirable property that determines how good and appropriate it is for the task at hand (Michie et al., 1994). For generality, we define a *predictor* as a classifier for classification or regression problems.

Instance An instance (also described as an example, case or record in the literature), is a fixed list of features values (or a vector of feature values). An instance describes the basic entity that we are dealing with, such as a patient, a DNA sequence or a medical data about some disease.

Feature A feature (sometimes called an attribute or field) describes some characteristic or aspect of an instance. Normally, there are two features type: nominal (*e.g.*, color: red, green, blue) and continuous (*e.g.*, weight $\in \mathcal{R}$, a real number). Continuous features are used whenever there is a linear ordering on the values, even if they are not truly continuous (*e.g.*, weight may be specified to the nearest integer kilogram).

For any sort of features, usually there is also an important value that means *unknown* for most inducers. This special value is very different of, for instance, zero-values (for

numbers) or empty strings. In most of the well known inducers this value is represented as a question mark (**?**).

Also, several inducers assume that the original features describing instances are *relevant* enough to learn the task at hand. However, some of the features may not be directly relevant and other features may be *irrelevant*. A feature is irrelevant if there is a *complete* and a *consistent* description (see definition below) of the classes to be learned that does not use that feature (Michalski and Kaufman, 1998). Thus, a *nonessential* feature may be either relevant or irrelevant but it can be dispensable when learning takes place (Lee, 2000).

One important issue to be considered is choosing features with predictive capability. No matter what learning method is employed, the concepts which can be learned are at the mercy of the data and the quality of the features (Weiss and Kulikowski, 1991). For example, one could chose features with low predictive power, such as (hair color, eyes color, car model, number of children) or features with higher predictive power, such as (temperature, skin resistance, lung examination). For this specific learn task, in the second case, better predictions based on new unlabeled instances probably will occur than in the former case .

Class In supervised learning every instance has a special feature, the label (also called output or class), which describes the phenomenon of interest *i.e.*, the task we would like to learn and make predictions about. An unlabeled instance is the part of the instance without the label *i.e.*, the list of feature values. The class label values are typically drawn from a discrete (nominal) set of classes $\{C_1, C_2, \ldots, C_k\}$ in the case of *classification* or from the real values in the case of *regression*.

Dataset A dataset is a set of classified (labeled) instances. Table 1 shows the general format of a dataset T with n instances and m features. In this table a row i refers to the i-th instance ($i = 1, 2, \ldots, n$) and column entries x_{ij} refer to the value of the j-th ($j = 1, 2, \ldots, m$) feature X_j of instance i.

	X_1	X_2	$\cdots$	X_m	Y
T_1	x_{11}	x_{12}	$\cdots$	x_{1m}	y_1
T_2	x_{21}	x_{22}	$\cdots$	x_{2m}	y_2
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
T_n	x_{n1}	x_{n2}	$\ldots$	x_{nm}	y_n

Table 1: Dataset in the feature-value or spreadsheet format

As can be seen, instances are tuples $T_i = (x_{i1}, x_{i2}, \ldots, x_{im}, y_i) = (\mathbf{x}_i, y_i)$ also referred as ($\mathbf{X}$, class($\mathbf{X}$)) or $(\mathbf{X}, Y)$ where the last column, Y, is what we try to predict given the other $\mathbf{X}$ features *i.e.*, $Y = f(\mathbf{X})$. Observe that each $\mathbf{X}$ is an element of the set $X_1 \times X_2 \times \ldots \times X_m$ where X_j is the domain of the j-th feature and Y belongs to one of the k classes *i.e.*, $Y \in \{C_1, C_2, \ldots, C_k\}$.

Usually, a dataset is split into two disjunct sets: the *training set* which is used to learn the concept and the *test set* used to measure the effectiveness of the learned concept. They are normally disjunct to ensure that measures using the test set are from an independent set yielding a statistically valid measurement.

Note that, after inducing a classifier, it is possible to evaluate this classifier in the training set as well as in the test set. It is usual to call measurements using the performance of

the classifier in the training set as *apparent* (also know as *resubstitution*) measure and in the test set as *true* measure (like, for example, *apparent error* and *true error*). For most type of predictors, the apparent measure is a poor estimator of future performance, since it tends to be biased optimistically.

Noise It is common, in the real world, to work with imperfect data. Imperfect data could be derived from the process which generates the data itself, the data acquisition process, the data transformation process or even incorrectly labeled classes (for example, instances with the same feature value but different class labels). In this case, we say there is *noise* in the data. Also, features that are not more predictive than chance to the learning task can be considered noise (Weiss and Kulikowski, 1991, Chapter 1).

Data Preparation Usually, there are several steps preceding the learning step. For example a cleaning step can be used to improve the quality of data by modifying its form or content, such as removing or correcting data values that are incorrect. If this step is not initially applied to the data, the learning process may indicate that further cleaning is desired for improving the quality of the extracted knowledge.

After cleaning, it may be still necessary to transform the data before passing it trough an inducer. This may require smoothing feature values or even data transformations in order to meet the requirements imposed by specific inducers.

Also, if there is a lot of nonessential features in the initial set of instances, the complexity of the learning process may increase significantly. In this case, data reduction techniques can be used. One possible approach is to use Feature Subset Selection — FSS — methods to determine the most relevant features for a given inducer (Kira and Rendell, 1992; John et al., 1994; Blum and Langley, 1997) . In (Baranauskas and Monard, 1998b; Baranauskas and Monard, 1998a; Baranauskas et al., 1999a; Baranauskas et al., 1999b; Baranauskas and Monard, 1999; Lee et al., 1999) we report some experimental results using FSS by filtering and wrapping inducers around (Kohavi and Sommerfield, 1995; Kohavi, 1997). Refer to (Weiss and Indurkhya, 1998) for a good review of data preparation techniques.

Background Knowledge In general Background Knowledge — BK — includes information about valid feature values, a preference criterion for choosing among possible features or even hypotheses. BK may also include relationship constraints between features, rules for generating high level concepts, new features possibly derived from the original ones, as well as some initial hypothesis. Not all inducers are able to use BK when learning concepts *i.e.*, they use only the data. Observe that the number of hidden units and weights as well as topology in a neural net is a sort of background knowledge provided by the user.

Classifier Given a set of training instances, the inducer outputs a *classifier* (also called a *hypothesis* or a *concept description*) such that, given a new instance, it accurately predicts its label Y. All classifiers use stored data structures that are then interpreted as a mapping for an unclassified instance to a label.

Formally, in supervised classification an instance is a pair $(\mathbf{X}, f(\mathbf{X}))$ where $\mathbf{X}$ is the input and $f(\mathbf{X})$ is the output. The task of an inducer is, given a set of instances, to induce a function h that approximates f. In this case, h is called an *hypothesis* over f.

Figure 3 shows an example of this concept (Russel and Norvig, 1994). Suppose we have as instances points in the plane shown as circles in (a), where the task is to find a function $h(\mathbf{X})$ that fits the points well. A possible approximation for the real function f could be to connect each instance with the next through straight lines (b). Of course, this could be

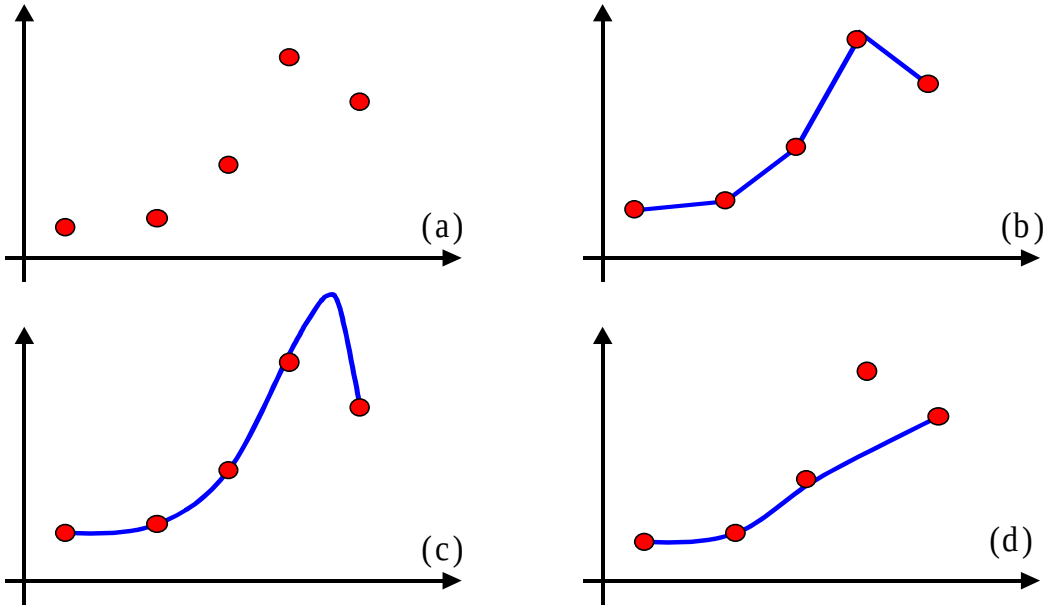


Figure 3: Given the instances in (a), represented as points in the form $(\mathbf{X}, f(\mathbf{X}))$, (b), (c) and (d) show possible consistent hypotheses h for approximating the real function f which is unknown

a over simplified approximation for function f. So, we could approximate f by a smooth polynomial function, as can be seen in (c). However, this hypothesis could be considered too complicated by the final user. Hence, a more simple hypothesis could be induced as shown in (d), which completely ignores one instance *i.e.*, treating outliers as noise.

Bias Still considering Figure 3, as the real function f is actually unknown there are many possible choices for h, but without extra knowledge, we have no way to choose between hypotheses (b), (c) or (d). Any preference for one hypothesis over another, besides simple consistency with the instances, is called a *bias* (Russel and Norvig, 1994). Because there are almost always a large number of possible consistent hypotheses all inducers exhibit some sort of bias. Indeed, learning without bias is impossible. Therefore, an unbiased predictor means that the predictor will, over a long run, average to the true error.

Variance Besides bias, another important factor to be aware of is the *variance* of the classifier, which measures how much the learning algorithm's guess fluctuates for different training sets of the given size (how often they disagree). See Section 2.6, page 21, for more details.

Description Space It can be observed that m features can be interpreted as a vector, each feature corresponding to a coordinate in a m-dimensional space (or *description space*). Also, each point in the space can be labeled with its corresponding associated class.

From this point of view, a classifier divides this space into regions, each region labeled with a class. An unseen instance is classified by determining the region where the corresponding labeled point falls into and assigning the class associated with that region.

Figure 4 shows an example for two classes {o, +} and two features $\{X1, X2\}$. Suppose in (a) the extracted classifier, in symbolic form, is given by '**if** $X1 < 5$ **and** $X2 < 8$ **then** class o **else** class +'. This classifier divides the description space into a two regions: the inside of the rectangle where $X1 < 5$ and $X2 < 8$ (considering positive values for both

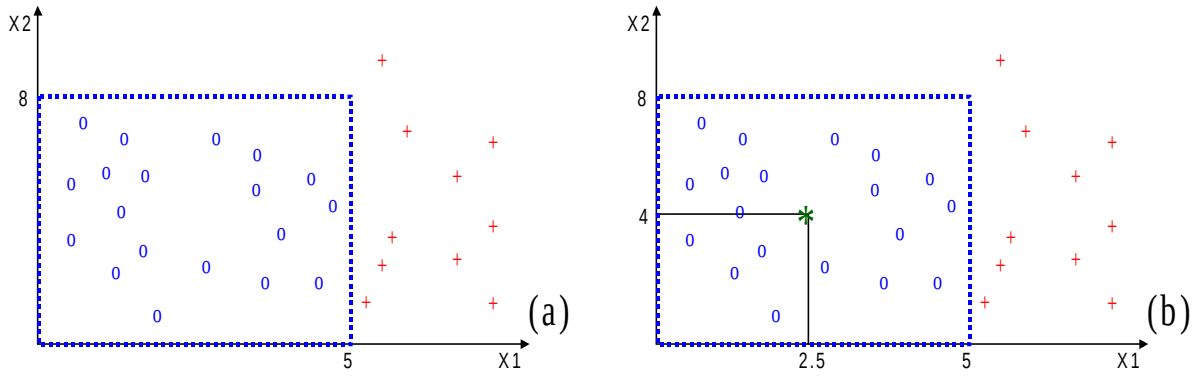


Figure 4: A classifier divides the description space into regions, each region labeled with a class (a); the unlabeled instance '*' is classified according with the region where it falls (b)

features) and outside it. Now, in (b) a new unlabeled instance '*' with $(X1, X2) = (2.5, 4)$ will be classified as class o since it falls inside the region which defines that class.

Stability Some inducers are *unstable* in the sense that small perturbations in the training set or in its construction may result in large changes in the extracted predictor. For instance, feature subset selection in regression, decision trees in regression or classification and neural nets are all unstable; nearest neighbors and linear discriminant analysis are stable (Breiman, 1996d).

Unstable predictors are characterized by high variance: as the training set changes, extracted predictors can differ considerably from each other. *Stable predictors*, on the other hand, do not change much over small changes in the training set. There is a trade-off between bias and variance: unstable predictors usually have high variance but they can have low bias. Stable predictors have low variance, but they can have high bias.

Error and Accuracy A very common measure used is the *error rate* of a classifier h, also know as misclassification rate denoted by $\mathsf{ce}(h)$ in classification problems. Usually the error rate is obtained using Equation 1 which compares each labeled instance with the predicted classifier label. The $\parallel E \parallel$ operator returns 1 if E is true and zero otherwise and n is the number of instances. The error rate complement, the classifier *accuracy*, denoted by $\mathsf{ca}(h)$ is given by Equation 2.

$$\mathsf{ce}(h) = \frac{1}{n} \sum_{i=1}^{n} \parallel y_i \neq h(x_i) \parallel \tag{1}$$

$$\mathsf{ca}(h) = 1 - \mathsf{ce}(f) \tag{2}$$

For regression problems, the *predictor error* (pe) can be estimated calculating the distance between the true value against the predicted one. Usually, two measures are commonly used: the *mean squared error* (mse) and the *mean absolute distance* (mad), given by Equations 3 and 4, respectively.

$$\text{pe-mse}(h) = \frac{1}{n}\sum_{i=1}^{n}(y_i - h(x_i))^2 \tag{3}$$

$$\text{pe-mad}(h) = \frac{1}{n}\sum_{i=1}^{n}|y_i - h(x_i)| \tag{4}$$

Under- and Overfitting When extracting an hypothesis from data, it is possible the predictor to be very specific for the training set. As the training set is only a sample of all possible instances, it is possible to generate predictors that improve performance on this training set while decreasing performance on other instances outside this training set. In this situation the error (or other measure) on an independent test set yields to a poor performance predictor (Weiss and Indurkhya, 1998). In this case we say that the predictor *overfits* the training set.

For example, the best classifier is the Bayes classifier which is equivalent to a direct table lookup for instances in the training set. This table lookup classifier have zero apparent error rate since testing the original data makes no mistakes (assuming instances with same feature values belonging to the same class *i.e.*, noise-free instances). But when using new data it will be extremely difficult to obtain zero error due to the enormous number of possible combinations.

On the other hand, it is also possible that few representative instances are given to the learning system (for example, decision trees or rule induction algorithms) or the user pre-defines the classifier size too small (for example, insufficient neurons and weights are defined in a neural net or a high pruning factor is defined for a decision tree) or a combination of both. Therefore, it is also possible to generate predictors that poorly improve performance on the training set as well as on a test set. In this case, the predictor *underfits* the training set.

Figure 5 illustrates the impact of under- and overfitting. The horizontal axis of this figure indicates the complexity (size) of the predictor (*i.e.* number of nodes in a decision tree, number of rules in rule induction or number of neurons or weights in a neural net). The vertical axis indicates the error of predictions made by the predictor. The dotted line shows the error of the predictor over the training set whereas the solid line shows error measured over the test set which is not included in the training set. As expected, the error over the training set (apparent error) decreases monotonically as the predictor is built. However, the error measured over the test set (true error) first decreases up to $N2$ then increases.

Having this in mind, which hypothesis should be better? The *Ockham's razor* (Kearns and Vazirani, 1994) principle gives a clue. It states that *the most likely hypothesis is the simplest one that is consistent with all observations.* In fact, this means that, other things being equal, a simple hypothesis that is consistent with the instances is more likely to be correct than a complex one.

Overtuning Overtuning may occur when an algorithm developer tunes a learning algorithm, or its parameter settings, too well in order to optimize its performance on all the available data. This may be a harmful technique whenever all available data is used for algorithm development and tuning. Like overfitting, overtuning can be detected and avoided by

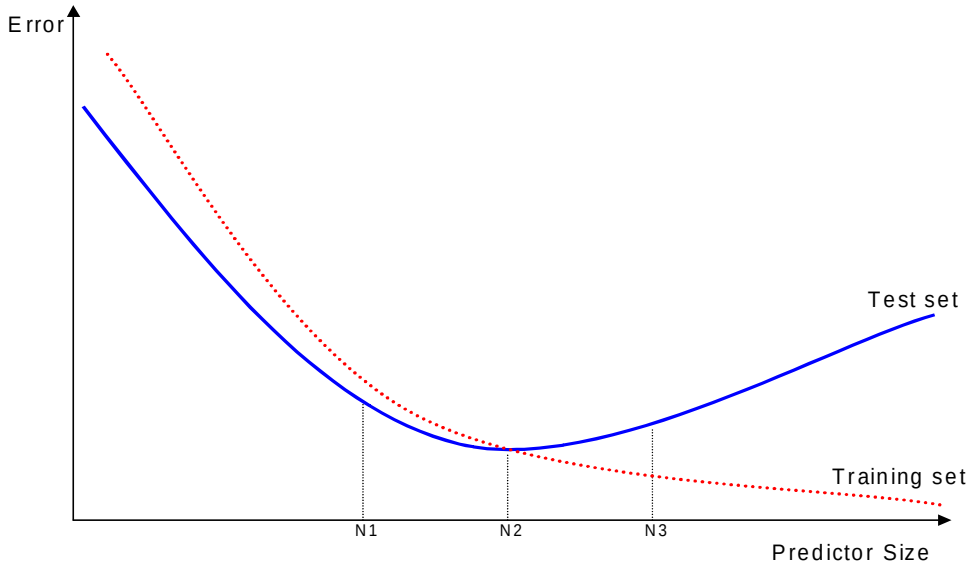


Figure 5: The relationship between predictor size and error

using only part of the data for algorithm development and using the remainder of the data for final system testing. Of course, as ordinary overfitting, if the data is not held out for final evaluation, the observed performance of the system cannot be confidently used as an estimate of the expected performance of the system.

Pruning Pruning is a standard way of dealing with noise and overfitting in decision trees or rule learning. The common idea is to deal with the problem of overfitting by learning an overly general hypothesis from the training set in order to improve the prediction on unseen instances. There are, basically, two different approaches to pruning:

1. *pre-pruning* which means that during the hypothesis generation some training instances are deliberately ignored in order that the final hypothesis does not classify all training instances correctly;
2. *pros-pruning* which means that first the hypothesis that perfectly explains all training instances is generated. After that, the hypothesis is generalized by eliminating some parts like cutting off branches of decision trees or some conditions in rule induction.

Completeness and Consistency Once induced, an hypothesis can be evaluated about its *completeness*, if it classifies all instances and *consistency*, if it correctly classifies the instances. So, given an hypothesis, it can be considered: (a) complete and consistent; (b) incomplete and consistent; (c) complete and inconsistent or (d) incomplete and inconsistent. Figure 6 shows an example of these four cases considering two features X_1 and X_2, three class labels (o, +, *) and the induced hypothesis for each class. They are represented by two solid line regions for class o, two dotted line regions for class + and one dashed line region for class *.

Complex A <complex> is a disjunction of conjunctions of feature tests in the form:

$$X_i \ op \ Value$$

where X_i is a feature, *op* is an operator in the set $\{=, \neq, <, \leq, >, \geq\}$ and *Value* is a valid feature X_i constant value.

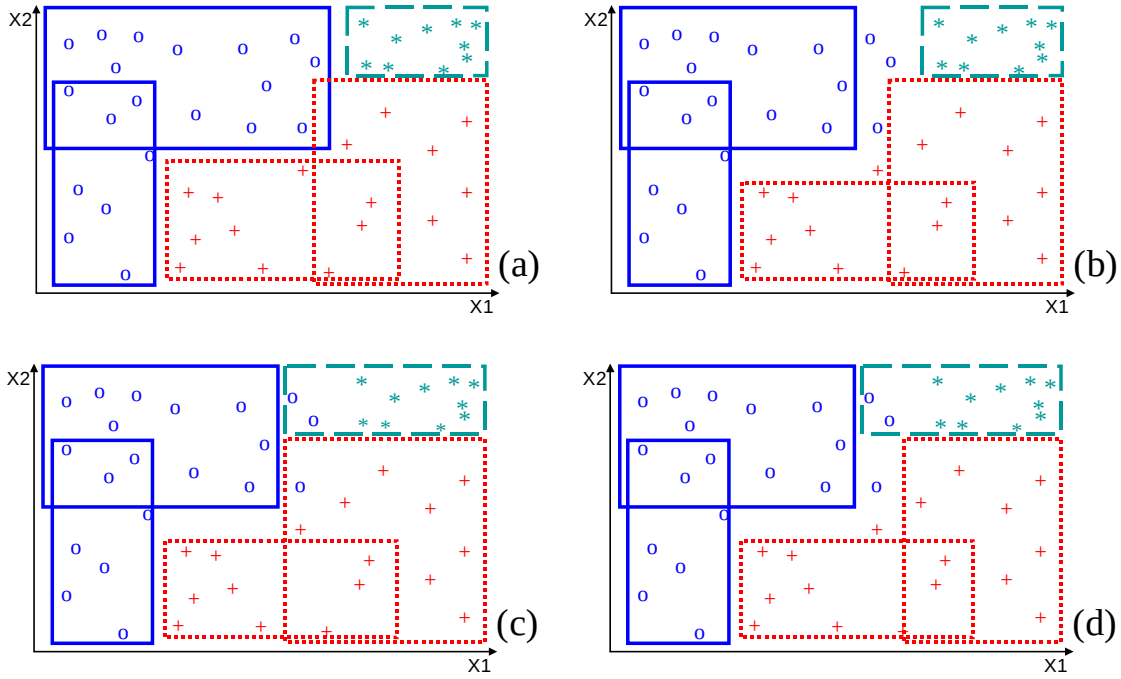


Figure 6: Hypothesis completeness and consistency

It is also possible to have a linear combination of features (for continuous features) in the form:

$$c_1 \times X_1 + c_2 \times X_2 + \ldots + c_m \times X_m \; op \; Value$$

where c_i is a constant, X_i is a continuous (integer or real) feature, *op* is an operator in the set $\{<, \leq, >, \geq\}$ and *Value* is a constant value.

Rule A rule assumes the form

$$\textbf{if} \; <\text{complex}> \; \textbf{then} \; <\text{class} = C_i>$$

where C_i belongs to the set of possible k class values $\{C_1, C_2, \ldots, C_k\}$. The <complex> part is also denominated rule *conditions* and <class $= C_i$> is denominated rule *conclusion*. Learned rules are usually *consistent* and *complete* with regard to the training data.

Instances that satisfy the <complex> part of the rule composes its *covered set* or, in other words, those instances are *covered* by the rule. Instances that satisfy both the <complex> and the conclusion <class $= C_i$> are *positively covered* by the rule. On the other hand, instances satisfying the <complex> but with <class $\neq C_i$> are called *negatively covered*, as shown in Table 2.

Instances satisfying ...	Are ...
<complex>	covered by rule
<complex> **and** <class $= C_i$>	positively covered by rule
<complex> **and** <class $\neq C_i$>	negatively covered by rule

Table 2: Rule coverage definitions

Association Rule Association learning systems find conjunctive implication rules of the form:

$$\textbf{if } <\text{complex}_1> \textbf{ then } <\text{complex}_2>$$

where there are no common features among them, *i.e.* $<\text{complex}_1> \cap <\text{complex}_2> = \emptyset$. As can be noted, there is no explicit class definition in this case and any feature can be used as part of rule conclusions. For instance:

$$\textbf{if } X_3 \textbf{ and } X_5 \textbf{ then } X_1 \textbf{ and } X_2$$

Observe that an association rule is a generalization of a conventional rule defined previously. Thus, criteria used to evaluate association rules can be used to evaluate conventional rules as well. Conventional association systems find association rules that satisfy some minimum criteria (Agrawal et al., 1993; Agrawal and Srikant, 1994; Klemettinen et al., 1994; Bayardo and Agrawal, 1999). Therefore, it is easy to extend rule coverage definitions shown in Table 2 to association rules as Table 3 shows.

Instances satisfying ...	Are ...
$<\text{complex}_1>$	covered by rule
$<\text{complex}_1>$ **and** $<\text{complex}_2>$	positively covered by rule
$<\text{complex}_1>$ **and not** $<\text{complex}_2>$	negatively covered by rule

Table 3: Association rule coverage definitions

Two widely used criteria for rule evaluation are *support* and *confidence*. Let us denote n as the total number of training instances, n_1 as the number of instances satisfying $<\text{complex}_1>$ and n_{12} as the number of instances that satisfy both $<\text{complex}_1>$ and $<\text{complex}_2>$. Thus, Equations 5 and 6 define *support* and *confidence*, respectively.

$$\text{Support} = \frac{n_{12}}{n} \tag{5}$$

$$\text{Confidence} = \frac{n_{12}}{n_1} \tag{6}$$

Confusion Matrix A confusion matrix offers a effectiveness measure of the classification model *i.e.*, the induced hypothesis h, by showing the number of correct classifications against predicted classifications for each class. Results are summarized in two dimensions, namely true classes and predicted classes as shown in Table 4 for k different classes $\{C_1, C_2, \ldots, C_k\}$. Each element $M(C_i, C_j)$ of this matrix indicates the number of instances that actually belongs to the true class C_i but were predicted as being class C_j as shown in Equation 7 where T represents the dataset. As defined before the $\parallel E \parallel$ operator returns 1 if E is true and zero otherwise.

$$M(C_i, C_j) = \sum_{\{\forall (x,y) \in T : y = C_i\}} \parallel h(x) = C_j \parallel \tag{7}$$

Class Label	predicted C_1	predicted C_2	$\cdots$	predicted C_k
true C_1	$M(C_1, C_1)$	$M(C_1, C_2)$	$\cdots$	$M(C_1, C_k)$
true C_2	$M(C_2, C_1)$	$M(C_2, C_2)$	$\cdots$	$M(C_2, C_k)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
true C_k	$M(C_k, C_1)$	$M(C_k, C_2)$	$\cdots$	$M(C_k, C_k)$

Table 4: Confusion matrix

The number of correct predictions for each class falls along the main diagonal $M(C_i, C_i)$ of the matrix . All other elements $M(C_i, C_j)$, for $i \neq j$ are the number of errors for a particular type of misclassification error.

Of course, the ideal classifier would have all these entries equal to zero since it makes no mistakes, as Table 5 shows.

Class Label	predicted C_1	predicted C_2	$\cdots$	predicted C_k
true C_1	$M(C_1, C_1)$	0	$\cdots$	0
true C_2	0	$M(C_2, C_2)$	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
true C_k	0	0	$\cdots$	$M(C_k, C_k)$

Table 5: Confusion matrix for ideal classifier

Given a rule, an instance and a class, there are four possible cases that might occur:

1. the instance satisfies the <complex> (all conditions) of the rule and its class is the same as the one predicted by the <class $= C_i$> conclusion.
2. the instance satisfies the <complex> (all conditions) of the rule and its class is **not** the same as the one predicted by the <class $= C_i$> conclusion.
3. the instance **does not** satisfy the <complex> (all conditions) of the rule and its class is the same as the one predicted by the <class $= C_i$> conclusion.
4. the instance **does not** satisfy the <complex> (all conditions) of the rule and its class is **not** the same as the one predicted by the <class $= C_i$> conclusion.

The error rate $\mathsf{ce}(h)$ and its complement accuracy $\mathsf{ca}(h)$ — defined in Equations 1 and 2, respectively in page 9 — are two of the more widely used metrics for evaluating performance of learning systems. For simplicity let us consider a two-class classification problem. With just two classes, usually labeled as "+" and "−", the choices are structured to predict the occurrence or non-occurrence of a single event or hypothesis. When just two classes are considered, the two possible errors are denominated *false positive* and *false negative*. Table 6 illustrates the confusion matrix for two-class classification problem where T_P is the number of correctly classified positive examples and F_N is the number of misclassified positive instances from a total of $n = (T_P + F_N + F_P + T_N)$ instances.

Still considering Table 6, note that four situations may occur, illustrated in Figure 7 for <class $= C_+$>.

1. the instance belongs to class C_+ and is predicted by the classifier as class C_+. In this case, we say this instance is a *true positive*.
2. the instance belongs to class C_- and is predicted by the classifier as class C_-. In this case, we say this instance is a *true negative*.

Class label	predicted C_+	predicted C_-	Class error rate	Total error rate
	True positives	False negative		
true C_+	T_P	F_N	$\frac{F_N}{T_P+F_N}$	$\frac{F_P+F_N}{n}$
	False positives	True negatives		
true C_-	F_P	T_N	$\frac{F_P}{F_P+T_N}$	

Table 6: Two class classification performance

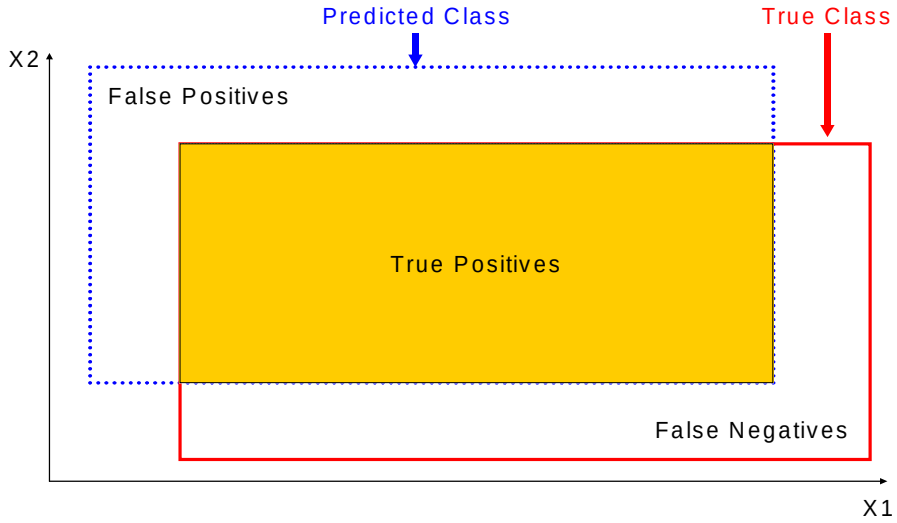


Figure 7: True positives, false positives and false negatives

3. the instance belongs to class C_- and is predicted by the classifier as class C_+. In this case, we say this instance is a *false positive*.
4. the instance belongs to class C_+ and is predicted by the classifier as class C_-. In this case, we say this instance is a *false negative*.

Other frequency ratios derived from the numbers in Table 6 can be computed as illustrated in Equations 8 to 12 (Weiss and Kulikowski, 1991).

$$C_+ \text{ Predictive Value} = \frac{T_P}{T_P + F_P} \tag{8}$$

$$C_- \text{ Predictive Value} = \frac{T_N}{T_N + F_N} \tag{9}$$

$$\text{True } C_+ \text{ Rate or Sensitivity or Recall} = \frac{T_P}{T_P + F_N} \tag{10}$$

$$\text{True } C_- \text{ Rate or Specificity} = \frac{T_N}{F_P + T_N} \tag{11}$$

$$\text{Accuracy} = \frac{T_P + T_N}{n} \tag{12}$$

For example, high sensitivity (or low error rate in C_+) indicates the ability to correctly classify positive instances. However, the specificity may be poor if many positive instances are incorrectly classified as negative.

Error Costs Properly measuring the performance of classifiers, through error (or accuracy) rate, plays an important role in Machine Learning, since the goal is building classifiers with low error rate on unseen instances (Batista, 1997; Batista and Monard, 1998). Still, considering the two-class situation, if the cost of having false positive error and false negative error is not the same, then other performance measures should be used. A natural alternative is to use *misclassification cost* which is a penalty number assigned for making a mistake. In this case errors are converted into costs by multiplying an error by its misclassification cost. Thus, instead of designing a classifier to minimize error rates, the goal would be to minimize misclassification costs.

Class Prevalence Another fairly common issue is the imbalance of the dataset's class prevalence. For example, suppose a dataset with class distribution — the proportion of instances in each class — $(C_1, C_2, C_3) = (99.00\%, 0.25\%, 0.75\%)$. A simple classifier that predicts always the majority class C_1 would have an accuracy of 99.00%. This can be undesirable when minority classes are those which hold very important information, for example, predicting C_1: normal patients, C_2: patient has disease A and C_3: patient has disease B.

It is important to be aware, when working with imbalanced datasets, that is desirable to use a performance measure other than accuracy (Kubat et al., 1998b). This is due to the fact that most learning systems are designed to optimize accuracy, behaving poorly if the training set is highly imbalanced. The induced classifiers tend to be highly accurate on the majority class instances but usually misclassify many of minority class instances. Some approaches have been developed to deal with this issue like a mechanism that removes redundant or harmful instances or a method to detect borderline and noisy instances (Kubat and Matwin, 1997; Kubat et al., 1997; Batista et al., 2000).

Learning Mode Whenever all instances should be present for learning, the learning mode is *non–incremental* also known as *batch* mode. On the other hand if the inducer does not have to start from scratch when new instances are added to the training set the mode is *incremental*. Therefore, in the incremental mode the inducer just try to update its old hypothesis whenever new instances are added to the training set.

In general, non–incremental learning should give better results since the inducer is allowed to look at all training instances at once. However, if computational time is an important issue and new instances are frequently added to the training set, non–incremental learning could save time.

2.4 Description Languages

When solving problems using computers, it is important to define how to translate the problem into computational terms. Specifically, in ML this means how to represent instances, hypothesis and the background knowledge. To describe them, the following *description languages* are used:

- Instance Description Language — IDL;
- Hypothesis Description Language — HDL;
- Background Knowledge Description Language — BDL

We outline some description languages most frequently used in learning in ascending order of their complexity and expressive power. We provide just intuitional explanations of these languages, avoiding unnecessary mathematical complexity. Since a description language can represent instances, hypotheses or background knowledge within this section we refer to them simply as an *object*.

Zero Order or Propositional Calculus In propositional calculus the object to be represented is described by conjunctions, disjunctions and negations of boolean constants that stand for individual features. For example:

$$\mathsf{female} \wedge \mathsf{grown\_up} \rightarrow \mathsf{can\_have\_children}$$

This language has low descriptive power, being not able of describing objects where relations are involved.

Attributional Logic In order to represent instances and concepts many of the existing propositional inductive learning algorithms use a *feature-based* (also called *attribute-based*) language. Formally, attributional logic is equivalent to propositional calculus but employes a more powerful and flexible notation. The improvement is that the features are treated as variables that can take on various values. For instance:

$$\mathsf{sex{=}female} \wedge \mathsf{age{=}grown\_up} \rightarrow \mathsf{class{=}can\_have\_children}$$

or equivalently,

$$\mathsf{sex(female)} \wedge \mathsf{age(grown\_up)} \rightarrow \mathsf{class(can\_have\_children)}$$

Although most inducers use attributional logic for describing instances and concepts, the strong constraint of this language still prevents representing structured objects as well as relations among objects or among its components. Thus, relevant aspects of the training instances, which somehow could characterize the concept being learned, cannot be represented.

First Order Logic In order to overcome the representational limitations imposed by an attributional language, learning in representations that are more powerful, such as some variants of first order logic has received more attention. First order logic provides a framework for describing and reasoning about *objects* and *predicates* that specifies *properties* or *relationships* among objects.

An important subset of first order logic is Horn clauses. A Horn clause consists of a head with just one predicate and a body with zero, one or more predicates. For example:

$$\mathsf{brother}(X,Y) \text{ :- } \mathsf{male}(X),\ \mathsf{parent}(X,Z),\ \mathsf{parent}(Y,Z)$$

This example says that a person X is brother of person Y if X is male and both X and Y have the same parent Z. The part to the left from the token :- is the head and the part to the right from the token is the body of the clause. The token :- is equivalent to the logical implication $\leftarrow$ and is called *neck*[1]. The commas separating each predicate stand for logical conjunctions. Also, all variables are always universally quantified. Variables inside parentheses are called *arguments* and the number of arguments is the predicate *arity*.

Observe that if all predicates have arity zero, the language reduces into zero order logic and if all predicates have arity one and all arguments are constants (no variables are involved), the language reduces into attributional logic.

Sets of first order Horn clauses can be interpreted as programs in the logic programming language Prolog (Bratko, 1990; Sterling and Shapiro, 1994). For this reason, inductive learning of first order rules is often called Inductive Logic Programming — ILP (Flach, 1994; Muggleton and Raedt, 1994; Caulkins, 1999).

Second Order Logic The second order logic is an extension of the first order logic, allowing the predicate names themselves to be considered as variables. For example, suppose the schema:

$$P_1(X,Y) \text{ :- } P_2(X),\ P_3(X,Z),\ P_4(Y,Z)$$

where P_1, P_2, P_3, P_4 are predicates. A possible instantiation could be

$$\mathsf{brother}(X,Y) \text{ :- } \mathsf{male}(X),\ \mathsf{parent}(X,Z),\ \mathsf{parent}(Y,Z)$$

Consequently, the schema remains untouched and only the predicate names can vary. Observe that this representational language is so rich and flexible that its use for is computationally infeasible. Sometimes, a common practice is to introduce constraints such as limiting the number of predicates in the clause, excluding recursive definitions or even limiting the number of predicate arguments (Morik et al., 1993).

Math Functions Math functions can be used to describe the hypotheses, specially for neural nets. In this case, the description space is divided into complex regions trough combinations of several math functions.

Table 7 shows the representational languages of some symbolic and non-symbolic inducers. Included in this table are decision tree systems such as CART (Breiman et al., 1984), and $\mathcal{C}4.5$ (Quinlan, 1988) as well as rule induction systems like $\mathcal{CN}2$ (Clark and Niblett, 1987; Clark and Niblett, 1989; Clark and Boswell, 1991), FOIL (Quinlan, 1990) and $\mathcal{R}ipper$ (Cohen, 1995). At last, one non-symbolic system, neural nets, is considered. Observe that only $\mathcal{R}ipper$ and FOIL are able to process background knowledge.

[1] q :- $p \equiv q \leftarrow p \equiv p \rightarrow q$

Inducer	IDL	HDL	BDL
$\mathcal{C}4.5$	attributional	attributional	
CART	attributional	attributional	
$\mathcal{CN}2$	attributional	attributional	
Ripper	attributional	attributional	attributional
FOIL	attributional	first order	first order
Neural Net	attributional	math functions	

Table 7: Description languages of some inducers

2.5 Searching

> Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.
>
> —*Winston Churchill*

When looking for good hypotheses an inducer can use several search strategies. A search process then explores *states* or *nodes* in the inducer's representational language description space according with:

Initial State The initial state is the starting point of the search.

Search Operators The search operators compose the set of possible actions that can be taken when searching. When one operator is applied to a state (or a set of states), the result is the set of states that can be reached by carrying out the actions in a particular case.

Termination Criterion The termination criterion, or goal test, is applied to a single state description to determine if it is the goal state.

Search Strategy The search strategy defines under what conditions and to which state an operator is to be applied.

Evaluation Function The evaluation function, used only in informed search, returns a number the describes the desirability of expanding a state.

Uninformed and informed search techniques are widely used in Machine Learning. Both are briefly described in the following two sections.

2.5.1 Uninformed, Exhaustive or Blind Search

In the uninformed search strategy the only information available is how to distinguish a goal state from a non goal state. Two fundamental strategies of uninformed search are following described.

Depth-first Given the initial state S_1, in the depth-first search an operator O_1 from the set $\{O_1, O_2, \dots\}$ is applied to it arriving at a new state S_2. If S_2 is not the goal state then operator O_1 is applied again to S_2, thus arriving at a new state S_3 and so on. If there are no more states to be reached from state S_3 then the search *backtracks*, returning to the

previous state S_2 and tries to apply another operator from the set $\{O_2, O_3, \dots\}$. If again it is not possible to reach any state from the present state the search backtracks further until a state that allows the application of some operator is found. If no such state can be found, the search terminates. Figure 8(a) shows the order in which the nodes are visited using depth-first search.

The depth-first search is neither *optimal* since it does not always find the highest quality solutions among several solutions nor *complete* because the strategy is not guaranteed to find a solution when there really is one. However, depth-first search has very modest memory requirements, since it needs to store only a single path from the initial state to the current state along with the remaining unexpanded sibling nodes for each on the path. Assuming b operators and d expansions — also called search *level* — performed so far, depth-first search requires $O(b^d)$ computer time but only $O(bd)$ of memory.

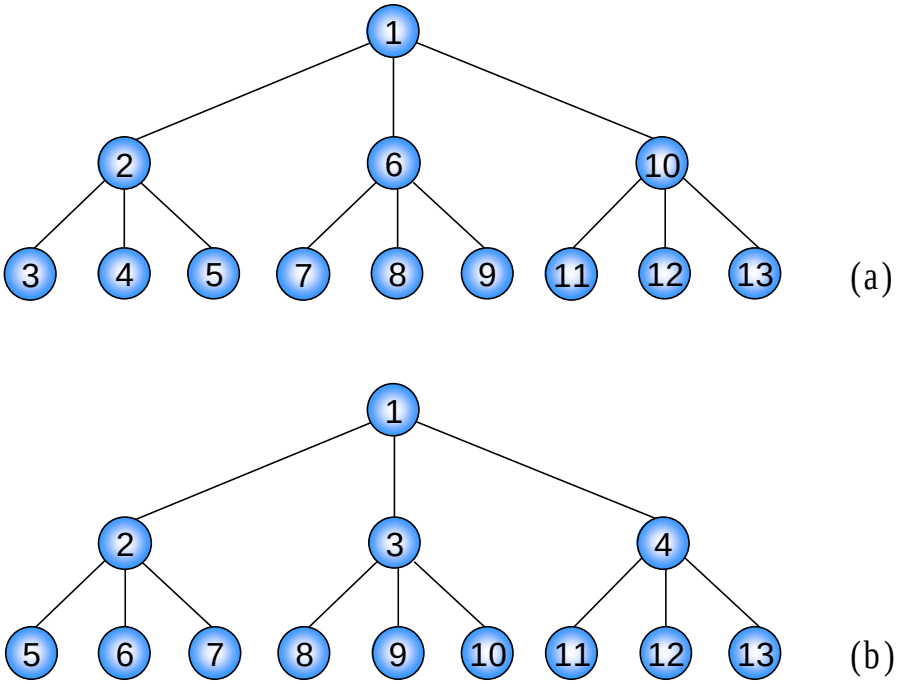


Figure 8: An example of depth-first search in (a) breath-first search in (b). The numbers inside each circle indicate the order in which states are visited

Breadth-first On the other hand, in the breadth-first search starting at the initial state S_1 all operators $\{O_1, O_2, \dots\}$ are applied to it, arriving at new states $\{S_2, S_3, \dots\}$. Then, each resulting state is tested using the termination criterion. If the termination test succeeds in some of them, the search terminates. Differently than depth-first, observe that breath-first search does not use backtracking. Figure 8(b) shows the order in which the nodes are visited using breadth-first search.

Contrary to depth-first search, breath-first search is optimal and complete. However, observe that at each state S_i all b operators are applied, thus given b new states. If the goal state is not reached, then b^2 new states are generated applying b operators to each one of the b previous states and so on.

Therefore, the number of states tested before finding a solution is $1 + b + b^2 + \ldots + b^d$, which includes the test in the initial state and d is the number of expansions performed so far. This is the maximum number *i.e.*, the worst case but note that the solution at

level d can be found at any point thus given, in the best case, a smaller number of states to be tested.

Two major problems about breath-first search is time and memory requirements both reaching $O(b^d)$, since all states must be maintained in memory at the same time. Considering $b = 10$ operators, $d = 10$ levels, and a computer able to process 1000 states/second, each state requiring 100 bytes of memory then 128 days and 1 Terabyte would be necessary for breath-first search to terminate (Russel and Norvig, 1994). Consequently, only in smaller problems breath-first search can be used.

2.5.2 Informed or Heuristic Search

As one can realize, uninformed search strategies are not very efficient in large search spaces. In these cases, *heuristics* guiding the search must be considered. Heuristics can decide which of the available operators will lead the search closest to the final state. As stated earlier this requires an evaluation function to assess the goodness value of each state reached.

Best-first The best-first search differs from breadth-first in that it always looks for the most promising state, trying to speed up the search. However, there is the danger of falling to a local maximum of the evaluation function rather than the global maximum.

Greedy For most problems, the evaluation function can be estimated but cannot be determined exactly. A function that estimates the evaluation function is called a *heuristic function*. When the best-first search uses the heuristic function to select the next state to expand it is called *greedy search*. For example, consider a map with several cities but no route is actually shown in the map. An heuristic function that can be used to replace the true distance from one city to others is the straight line distance.

Greedy search resembles depth-first search, suffering from the same defects: it is nor optimal neither complete.

Beam A more economical search than best-first (remember it stores all generated states) is the *beam search* that retains only S best states at any time. Therefore, at each step, if the number of current states is larger than S, the algorithm keeps only the S best states and deletes all others.

Hill-Climbing Hill-climbing is an instantiation of beam search, defining $S = 1$. The name is meant the resemblance to hill climbers striving to find the shortest path to the peak and always picking the steepest path.

2.6 The Bias plus Variance Decomposition

> This decomposition is useful in understanding the properties of predictors.
>
> —*Leo Breiman (Breiman, 1996c)*

The predictor *Fundamental Decomposition* (Breiman, 1996c; Breiman, 1996a) principle states that the predictor error can be viewed as three basic components:

1. the minimum error that can be obtained by the ideal classifier or predictor: the lower bound on the expected error of any learning algorithm;

2. the bias which measures how closely the learning algorithm's average guess, over all possible training sets of the given size that matches the target;

3. the variance which measures how much the learning algorithm's guesses will vary with respect to each other *i.e.*, how often it fluctuates for different training sets of the given size.

For classification, in Equation 13, $f = B^*$ is the Bayes classifier, which gives the minimum classification error rate $\mathsf{ce}(f) = \mathsf{ce}(B^*)$. For regression, in Equation 14, $\mathsf{pe}(f)$ is the minimum regression error given by the true hypothesis (function) f. Bias and variance are always positive terms — see (Breiman, 1996c) for more details on how to compute bias and variance of a predictor. At some data points bias predominates, at others the variance. But, in general, at each point $\mathbf{X}$ both contributions are positive.

$$\mathsf{ce}(h) = \mathsf{ce}(f) + \mathsf{bias}(h) + \mathsf{variance}(h) \tag{13}$$

$$\mathsf{pe}(h) = \mathsf{pe}(f) + \mathsf{bias}(h) + \mathsf{variance}(h) \tag{14}$$

This decomposition is important in comprehending the relationship between bias and variance and the behaviour of a predictor. In general, an inducer builds partitions in the description space in a certain way such that can be considered as a family of functions H. For instance, most of decision trees or rule induction inducers divide the space into rectangular regions whereas neural net can divide the space into more complex regions. In any case, each inducer tries to select the best classifier h, using the training set, from the set of functions H.

For example, if the family of functions H that can be generated by an inducer is the small set of linear functions and the true predictor f is fairly nonlinear, then the bias of h will be large. On the other hand, since a small number of parameters are estimated from the small set H, the variance of h will be low. But if H is a large family of functions like the set of functions represented by decision trees or neural nets then the bias is usually small (since is almost possible to approximate the true function f by some $h \in H$) but the variance can be large (because several parameters can be adjusted).

2.7 Measure Estimation: Resampling

> If you want to show that one algorithm is *always* more accurate, then forget it: this simply cannot be proven.
>
> —*Steven L. Salzberg (Salzberg, 1995)*

As described earlier, a learning system extracts a predictor from a dataset. Usually, the dataset itself is a *sample* from a larger population. We also have pointed out that the dataset is divided into two disjoint sets: the training set and the test set. Measures, for example errors, taken

from the test set can be considered as the true error since it approximates to the population error if the test set size is large enough. When test sample size reaches 1000, the estimates are extremely accurate and with 5000 instances the sample estimate is almost identical to the true error of the population (Weiss and Kulikowski, 1991).

It is important for estimating the true error the sample to be *random*. This means that the samples should not be pre-selected in any way. For real problems, one is given a sample from a single population of size n and the task is to estimate the true error for that population (not for all populations). There are several paradigms for estimating the true error that are following described and also summarized in Table 8.

	holdout	random	leave-one-out	r-fold cv	r-fold strat cv	bootstrap
Train size	pn	t	$n-1$	$n(r-1)/r$	$n(r-1)/r$	n
Test size	$(1-p)n$	$n-t$	1	n/r	n/r	$n-t$
Iterations	1	$I \ll n$	n	r	r	200
Replacement	no	no	no	no	no	yes
Class Prevalence	no	no	no	no	yes	yes/no

Table 8: Estimators Parameters

Resubstitution This procedure consists in building the classifier and testing its performance on the same dataset *i.e.*, the training set and the test set are identical. It is also called *apparent* measure estimation. It is well known that performances computed with this method are optimistically biased: the good performance on the training set does not extend to independent test sets.

Since the bias of the resubstitution estimator has been discovered, cross-validation methods have been proposed. They are all based on the same principle: there should be no common data in the learning and in the test sets.

Holdout The holdout estimator splits the data in a fixed percentage of instances p for training and $(1-p)$ for testing, usually taking $p > 1/2$. Typical values used are $p = 2/3$ and $(1-p) = 1/3$, although there is no theoretical foundations on these values.

In order to make the result less dependent on the splitting, one can average several holdout results, by building several partitions thus giving an averaged holdout method. Since a classifier designed on the entire data set will, on average, perform better than a classifier designed on only a part of the data, this method has a tendency to overestimate the actual error-rate.

Random Random subsampling can produce better error estimates than a holdout estimation. In this case, I predictors, $I \ll n$, are induced from each training set and the error is the average of the error for predictors derived for independently and randomly generated test sets.

r-fold cross-validation This estimator is a compromise between the holdout and the leave-one-out estimator. In the r-fold cross-validation — CV — the instances are randomly divided into r mutually exclusive partitions (folds) of approximately equal size of n/r instances, with $r \ll n$. The instances in the $(r-1)$ folds are independently used for training and the extracted predictor is tested on the remaining fold. This process is repeated r times, each time considering a different fold for testing. The cross-validation error is the average error over all r folds.

This rotation procedure reduces both the bias inherent to the holdout method and the computational burden of the leave-one-out. However, observe that, for example, in a ten-fold cross-validation, each pair of training sets shares 80% of the instances. It is easy to generalize that the proportion of shared instances in cross-validation is given by $(1 - 2/r)$ for $r \geq 2$ folds, plotted in Figure 9. As the number of folds increases, this overlap may prevent statistical tests from obtaining a good estimate of the amount of variation that would be observed if each training set were completely independent from each other.

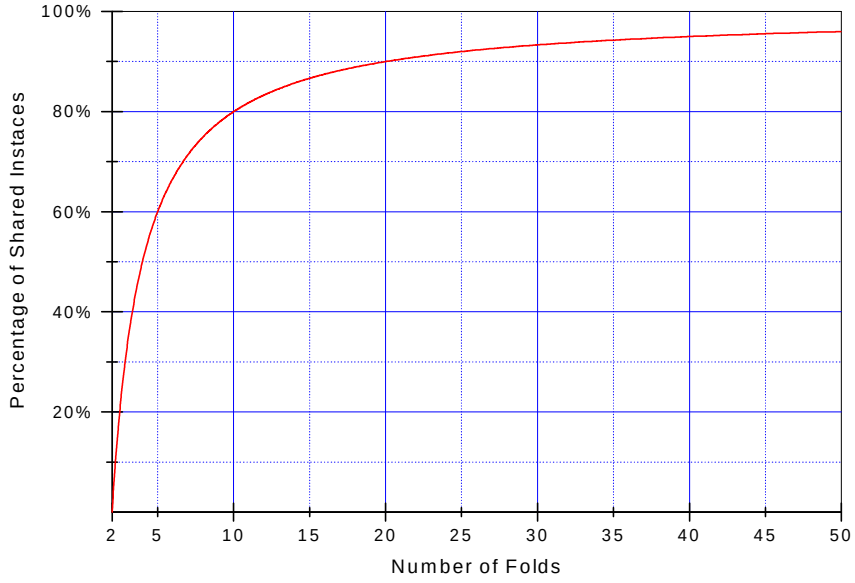


Figure 9: Number of folds versus percentage of shared training instances in cross-validation

r-fold stratified cross-validation The stratified cross-validation is similar to cross-validation but when generating the mutually exclusive folds, the class distribution — the proportion of instances in each class — is considered during sampling. This means, for instance, that if the original dataset with two class labels having 20% and 80% distribution, each fold will also follow this ratio.

Leave-one-out Leave-one-out is a special case of cross–validation. It is computationally expensive and is often used in small samples. For a sample of size n a predictor is extracted using $(n - 1)$ instances and tested on the single remaining instance. This process is repeated n times, each time building a predictor by leaving one instance out. The error is the sum of errors on the single test instances divided by n.

Bootstrap For the bootstrap estimator (Efron and Tibshirani, 1993), the basic idea is to repeat several times the whole classification experiment a large number of times and to estimate quantities like bias from these replicate experiments, each experiment being proceed on the basis of a new training set obtained by resampling with replacement in the original dataset.

There are many bootstrap estimators. The most common bootstrap estimator is the *e0* bootstrap. A bootstrap training set consist of n instances (as the original dataset) sampled *with replacement* from the original dataset. This means that an instance T_i may

not appear in the bootstrap training set, some T_i may appear more than once. The remaining instances (those that did not appear in the bootstrap training set) are used as the test set. In the remaining of this work we will always refer to bootstrap as the *e0* bootstrap.

For a given bootstrap sample, an instance in the training set has probability $1-(1-1/n)^n$ of being selected at least once in the n times instances are randomly selected from the training set. For large n, this is approximately $1-1/e = 0.632$.

Therefore, for this technique, the average fraction of non repeated instances in the training set is 63.2% and the expected fraction of such instances in the test set is 36.8%. The estimated error is the average of the error over the number of iterations. About 200 iterations for bootstrap estimates are considered necessary to obtain a good estimate if the original dataset is small. The estimate of the error from this method is statistically equivalent to the leave-one-out error estimate.

2.8 Combining Predictors: Ensembles

> Ensembles are well-established as a method for obtaining highly accurate classifiers by combining less accurate ones. There are still many questions, however, about the best way to construct ensembles as well as issues about how to understand the decisions made by ensembles.
>
> —*Thomas G. Dietterich (Dietterich, 1997a)*

An ensemble consists of a set of individual predictors whose predictions are combined when predicting novel instances labels. Previous research has shown that an ensemble is often more accurate than any of the individual predictors in the ensemble *i.e.*, multiple classifiers have been shown to lead to improved predictive accuracy when classifying instances that are not among the training set. There is considerable diversity in the methods used to assemble the ensembles, including:

Stack In Stacking (Wolpert, 1992) the descriptions of the training instances are extended to include the results of classifying the instances with an initial selection of classifiers. The new descriptions are then analyzed to yield further classifiers, and so on.

Window Windowing (Quinlan, 1988) selects a subset of instances (a window) from the training set and generates a hypothesis from this subset. This hypothesis is then used to classify the remaining training instances *i.e.*, those that have not been included in the window. If there are misclassified instances, a selection of them is added to the initial window and a second hypothesis is constructed from the enlarged window. This cycle is repeated until a hypothesis built from the current window correctly classifies all training instances outside the window or the number of cycles exceeds a pre-defined value L as shown in Algorithm 1 (once again, the $\parallel E \parallel$ operator returns 1 if E is true and zero otherwise).

Bagg Bagging (Breiman, 1996b) — Bootstrap Aggregation — generates one bootstrap sample S_1 and induces an hypothesis h_1. This cycle is repeated L times for each bootstrap sample $S_2, S_3, \ldots, S_L$ inducing hypotheses $h_2, h_3, \ldots, h_L$, respectively. After that, all extracted classifiers are combined into the final classifier h^* using a majority vote as shown in Algorithm 2 *i.e.*, the most commonly predicted class by all $h_1, h_2, h_3, \ldots, h_L$ classifiers is the predicted class given by h^* as Equation 15 shows.

Algorithm 1 Windowing

Require: Instances: a set of n labeled instances $\{(x_i, y_i), i = 1, 2, \ldots, n\}$
Inducer: a learning algorithm
W: the initial window size, $W := \max\{0.2n, 2\sqrt{n}\}$ in (Quinlan, 1988)
I: the increment to window, $I := \max\{0.2W, 1\}$ in (Quinlan, 1988)

1: **procedure** window(n,Instances,Inducer,L,W,I)
2: window$_l$:= sample(W,Instances)
3: $L := (n - W)/I$ *// number of windows*
4: **for** $l := 1$ **to** L **do**
5: h_l := Inducer(window$_l$)
6: error$(h_l) := \sum_{x_i \notin \text{window}_l} \parallel h_l(x_i) = y_i \parallel$ *// error on instances outside the window*
7: **if** error$(h_l) = 0$ **then**
8: **exit loop** *// all instances correctly classified*
9: **end if**
10: window$_{l+1}$:= window$_l$ + *include almost* I *misclassified instances by* h_l *from* Instances
11: **end for**
12: $h^*(x) := h_l(x)$
13: **return** h^*

Algorithm 2 Bagging

Require: Instances: a set of n labeled instances $\{(x_i, y_i), i = 1, 2, \ldots, n\}$
Inducer: a learning algorithm
L: the number of bagging classifiers

1: **procedure** bagg(n,Instances,Inducer,L)
2: **for** $i := 1$ **to** n **do**
3: $p_i := 1/n$ *// initialize normalized weights*
4: **end for**
5: **for** $l := 1$ **to** L **do**
6: S_l := bootstrap_sample(n,p,Instances)
7: h_l := Inducer(S_l)
8: **end for**
9: $h^*(x) := \underset{y \in \{C_1, C_2, \ldots, C_k\}}{\arg\max} \sum_{l=1}^{L} \parallel h_l(x) = y \parallel$
10: **return** h^*

$$h^*(x) = \underset{y \in \{C_1, C_2, \ldots, C_k\}}{\arg\max} \sum_{l=1}^{L} \parallel h_l(x) = y \parallel \tag{15}$$

In regression problems, h^* is usually taken as the average value for all predictors as shown in Equation 16.

$$h^*(x) = \frac{1}{L} \sum_{l=1}^{L} h_l(x) \tag{16}$$

Bagging is most effective on unstable learning algorithms where small changes in the training set result in large changes in predictions.

Wagg Wagging — Weight Aggregation — is similar to bagging but it change the weights of instances in the training set instead of sampling. In fact, this method repeatedly perturbs the training set as bagging does but, instead of sampling from it, wagging adds Gaussian noise to each weight with mean zero and a fixed standard deviation, usually two. At each trial wagging starts with all weights equal and then noise is added to the weights and a classifier is induced (Bauer and Kohavi, 1999).

Boost Boosting was introduced by (Schapire, 1990) as a method for boosting the performance of a weak learning algorithm. After some improvements the AdaBoost — Adaptive Boosting — was introduced by (Freund and Schapire, 1995), sometimes called AdaBoost.M1 (Freund and Schapire, 1996).

The boosting algorithm, shown in Algorithm 3, generates classifiers sequentially while bagging can generate them in parallel. Each training instance has an associated weight. At every boosting iteration a classifier is built from the weighted training instances and each case is then reweighted according to whether or not it is misclassified.

When inducing the first classifier all instances are equiprobable *i.e.*, each instance have the same weight associated with it. After inducing this first classifier, boosting changes the weights of the training instances provided as input to the inducer based on the classifiers that were previously built. This cycle is repeated L times.

The final classifier is formed using a weighted voting scheme where the weight of each classifier depends on its performance on the training set used to build it as Equation 17 shows. In this equation, error(h_l) is the error for classifier h_l in its corresponding training set.

$$h^*(x) = \underset{y \in \{C_1, C_2, \ldots, C_k\}}{\arg\max} \sum_{l=1}^{L} \log\left(\frac{1 - \text{error}(h_l)}{\text{error}(h_l)}\right) \parallel h_l(x) = y \parallel \tag{17}$$

The AdaBoost algorithm requires an inducer whose error is bounded by a constant strictly less than 1/2 — see (Freund and Schapire, 1995) for more detail. Some implementations of boosting uses *resampling* because some inducers are unable to support weighted instances (Freund and Schapire, 1996). Some evidence show that *reweighting* works better in practice (Quinlan, 1996) since the inducer is given all instances (with weights) instead of just a sample.

There are many revisions to the AdaBoost algorithm. Two special revisions suggested by (Breiman, 1996c; Breiman, 1996a) are shown in Algorithm 4. If error(h_l), the error of classifier h_l, becomes equal or greater than 1/2, better results can be obtained by setting all weights w_i equal to $1/n$ and restarting. Also, if error(h_l) is equal zero, making the subsequent step undefined, again all weights are set equal to $1/n$.

Another implicit change is related to the use of *resampling* instead of *reweighting* in the algorithm. This is an important issue, since, for deterministic inducers like decision trees or rule induction, setting all weights equal to $1/n$ would generate the same classifier again.

In (Bauer and Kohavi, 1999), reweighting is used as suggested in the original AdaBoost algorithm, but when error$(h_l) \geq 1/2$ or error$(h_l) = 0$, a bootstrap sample is taken with weights equal to 1/2 for all instances and almost $L = 25$ classifiers are built; also precision problems are reported for weights and they have changed the original weight distribution to sum n instead summing 1.

Algorithm 3 Boosting (AdaBoost.M1)

Require: Instances: a set of n labeled instances $\{(x_i, y_i), i = 1, 2, \ldots, n\}$
Inducer: a learning algorithm accepting instances weighting
L: the number of boosting classifiers

1: **procedure** boost(n,Instances,Inducer,L)
2: **for all** $i := 1$ **to** n **do**
3: $w_i := 1/n$ *// initialize the weights*
4: **end for**
5: **for** $l := 1$ **to** L **do**
6: **for** $i := 1$ **to** n **do**
7: $p_i := w_i/(\sum_{j=1}^{n} w_j)$ *// normalize the weights*
8: **end for**
9: $h_l :=$ Inducer(Instances,p)
10: error$(h_l) := \sum_{i=1}^{n} p_i \parallel h_l(x_i) \neq y_i \parallel$ *// compute the hypothesis error*
11: **if** error$(h_l) > 1/2$ **then**
12: $L := l - 1$
13: **exit loop**
14: **end if**
15: $\beta_l :=$ error$(h_l)/(1-$error$(h_l))$
16: **for** $i := 1$ **to** n **do**
17: **if** $h_l(x_i) = y_i$ **then**
18: $w_i := w_i \beta_l$ *// compute new weights*
19: **end if**
20: **end for**
21: **end for**
22: $h^*(x) := \underset{y \in \{C_1, C_2, \ldots, C_k\}}{\arg\max} \sum_{l=1}^{L} \log(1/\beta_l) \parallel h_l(x) = y \parallel$
23: **return** h^*

Arc Adaptively Resample and Combine — Arcing — is a term defined in (Breiman, 1996a) to describe the family of algorithms the adaptively resample and combine, including AdaBoost which he calls arc-fs (in honor of Freund & Schapire) as the primary example of an arcing algorithm. Breiman also contrasts arcing with the P&C family (Perturb and Combine), of which bagging is the primary example.

The arc-x4 algorithm, shown in Algorithm 5 was described by Breiman as being as accurate as arc-fs (AdaBoost) without the weighting scheme used when building the final arc-fs classifier. Thus, the AdaBoost's strength is derived from the adaptive reweighting of instances and not from the final combination, as stated by (Breiman, 1996c):

> "Here is how arc-x4 was devised. After testing arc-fs I suspected that its success lay not in its specific form but in its adaptive resampling property, where increasing weight was placed on those cases more frequently misclassified. To check on this, I tried three simple update schemes for the probabilities p_i. In each, the update was of the form $(1 + x_i^a)$, and $a = 1, 2, 4$ was tested on the waveform data[2]. The last one did the best and became arc-x4. Higher values of a were not tested so further improvement is possible."

[2]The value x_i refers to the number of times the instance i was misclassified by the previous extracted classifiers. Note that the notation x_i in this quote has no relationship with the notation adopted in this work to describe instances.

Algorithm 4 Boosting (with revisions suggested by Breiman)

Require: Instances: a set of n labeled instances $\{(x_i, y_i), i = 1, 2, \ldots, n\}$
Inducer: a learning algorithm accepting instances weighting
L: the number of boosting classifiers

1: **procedure** boost(n,Instances,Inducer,L)
2: **for** $i := 1$ **to** n **do**
3: $w_i := 1/n$ *// initialize the weights*
4: **end for**
5: **for** $l := 1$ **to** L **do**
6: **for** $i := 1$ **to** n **do**
7: $p_i := w_i/(\sum_{j=1}^{n} w_j)$ *// normalize the weights*
8: **end for**
9: $S_l :=$ bootstrap_sample(n,p,Instances)
10: $h_l :=$ Inducer(S_l)
11: error(h_l) $:= \sum_{x_i \in S_l} p_i \parallel h_l(x_i) \neq y_i \parallel$ *// compute the hypothesis error over S_l*
12: **if** error(h_l) $\geq 1/2$ or error(h_l) $= 0$ **then**
13: **for** $i := 1$ **to** n **do**
14: $w_i := 1/n$
15: **end for**
16: **else**
17: $\beta_l :=$ error(h_l)/(1−error(h_l))
18: **for** $i := 1$ **to** n **do**
19: **if** $h_l(x_i) = y_i$ **then**
20: $w_i := w_i \beta_l$ *// compute new weights*
21: **end if**
22: **end for**
23: **end if**
24: **end for**
25: $h^*(x) := \underset{y \in \{C_1, C_2, \ldots, C_k\}}{\arg\max} \sum_{l=1}^{L} \log(1/\beta_l) \parallel h_l(x) = y \parallel$
26: **return** h^*

In general, bagging is almost always more accurate than a single classifier, but it is sometimes much less accurate than boosting. On the other hand, boosting can create ensembles that are less accurate than a single classifier. In some situation, boosting can overfit noisy datasets, thus decreasing its performance.

On the other hand, ensembles usually generate a large classifier, contrary as stated in the Ockham's razor. For example, in (Margineantu and Dietterich, 1997), using the Frey-Slater letter dataset (Blake et al., 1998) with 16 numeric features and 16000 instances, it is possible to achieve very good accuracy on the test set with 4000 instances by voting 200 classifiers. Including training and test sets, the dataset requires less than 700 Kbytes (in fact, less than 370 Kbytes without delimiters). However, each classifier requires 295 Kbytes of memory, so an ensemble of 200 classifiers requires 58 Mbytes, more than 85 times greater than the dataset (or 164 times the whole dataset without delimiters).

Algorithm 5 Arcing (arc-x4)

Require: Instances: a set of n labeled instances $\{(x_i, y_i), i = 1, 2, \ldots, n\}$
Inducer: a learning algorithm accepting instances weighting
L: the number of arcing classifiers

1: **procedure** bagg(n,Instances,Inducer,L)
2: **for all** $i := 1$ **to** n **do**
3: $p_i := 1/n$ *// initialize the weights*
4: $m_i := 0$ *// number of misclassification of instance i by classifiers $1, 2, \ldots, l$*
5: **end for**
6: **for** $l := 1$ **to** L **do**
7: $S_l :=$ bootstrap_sample(n,p,Instances)
8: $h_l :=$ Inducer(S_l)
9: **for** $i := 1$ **to** n **do**
10: $m_i := m_i + \parallel h_l(x_i) \neq y_i \parallel$ *// update misclassifications*
11: **end for**
12: **for** $i := 1$ **to** n **do**
13: $p_i := (1 + m_i^4)/(\sum_{j=1}^{n} (1 + m_j^4))$ *// compute new normalized weights*
14: **end for**
15: **end for**
16: $h^*(x) := \underset{y \in \{C_1, C_2, \ldots, C_k\}}{\arg\max} \sum_{l=1}^{L} \parallel h_l(x) = y \parallel$
17: **return** h^*

2.9 Evaluating Predictors

> Experimental machine learning research needs to scrutinize its approach to experimental design. If not done carefully, comparative studies of classification algorithms can easily result in statistically invalid conclusions.
>
> —*Steven L. Salzberg (Salzberg, 1995)*

Machine Learning is a powerful tool, but there is no algorithm that dominates all others for all problems. Thus, it is important to understand the strengths and limitations of different algorithms.

One approach that can work well in practice is to try different algorithms, estimate their accuracy, and pick the one with the highest estimated accuracy for a given domain. This method will work well as long as there are not too many algorithms being tried and when we have reason to believe that they are appropriate for the domain.

A standard evaluation methodology adapted from (Russel and Norvig, 1994, Chapter 18) is the following:

1. collect a large set of instances, all with correct feature values and class labels *i.e.*, noisy free instances for a specific domain. If 100% noisy free data is not possible, try to reduce instances with incorrect feature values or wrong class labels. As much as possible, try to choose features with predictive power somewhat above random chance to the problem being considered;

2. randomly divide this set into two disjoint sets: the training set and the test set (or use

one of the resampling estimators or ensembles described in the two earlier sections);

3. apply one or more inducers to the training set, obtaining one distinct hypothesis h for each one;
4. measure performance of the generated predictor h on the test set. Here it is possible to measure simply error (see Section 2.9.2 for comparing algorithms using error). Other aspects can also be considered like time for learning, size of the extracted predictor, rule quality (Freitas, 1998a; Freitas, 1998b; Horst, 1999), etc;
5. to study the efficiency and robustness of an inducer, repeat steps 2 to 4 for different training sets and sizes of training sets;
6. if you are proposing or adjusting an inducer, start again with step 1 to avoid evolving the algorithm to work well on just one dataset for a specific domain.

2.9.1 Calculating Mean and Standard Deviation using Resampling

Before comparing two algorithms, some additional equations will be necessary. We assume the use of cross–validation since it is a fairly used method in ML. However, any other resampling technique (except resubstitution) can be used besides cross–validation.

Given an algorithm A, for each fold i the error, denoted by $\mathsf{pe}(h_i)$, $i = 1, 2, \ldots, r$ is calculated. Then the final mean, variance and standard deviation for all folds are calculated by Equations 18, 19 and 20, respectively.

$$\mathsf{mean}(A) = \frac{1}{r}\sum_{i=1}^{r}\mathsf{pe}(h_i) \tag{18}$$

$$\mathsf{var}(A) = \frac{1}{r}\left[\frac{1}{r-1}\sum_{i=1}^{r}\left(\mathsf{pe}(h_i) - \mathsf{mean}(A)\right)^2\right] \tag{19}$$

$$\mathsf{sd}(A) = \sqrt{\mathsf{var}(A)} \tag{20}$$

Observe that the term $1/(r-1)$ in Equation 19 comes from the definition of unbiased variance estimator whereas the term $1/r$ comes from the Central Limit Theorem (remember that $\mathsf{pe}(h_i)$ is itself an average) for estimating variance of averages (Moses, 1986, Chapter 4).

To make this idea more clear, suppose running a ten-fold cross–validation *i.e.*, $r = 10$, for algorithm A with the following errors $(5.50, 11.40, 12.70, 5.20, 5.90, 11.30, 10.90, 11.20, 4.90, 11.00)$. Then Equations 18 and 20 become:

$$\mathsf{mean}(A) = \frac{90.00}{10} = 9.00$$

$$\mathsf{sd}(A) = \sqrt{\frac{1}{10(9)}90.30} = 1.00$$

In general, the error is represented as its mean followed by the token "±" followed by its standard deviation; in our example, the error is 9.00 ± 1.00.

It is important to note that most of ML programs that perform cross–validation already take care of these calculations, for example, the $\mathcal{MLC}$++ library (Kohavi et al., 1994; Kohavi et al., 1996; Felix et al., 1998) or the MineSet™ (Rathjens, 1996) tool.

2.9.2 Comparing Two Algorithms

When trying to compare two algorithms just looking to numbers, for example error rates for classification or error for regression, it is not easy to realize if one algorithm is better than the other. In many situations, to compare a pair of ML algorithms, the error (mean and standard deviation) obtained through r-fold stratified cross–validation is usually used (to maintain class distribution). In fact, many papers report error using ten-fold cross–validation or stratified cross–validation.

When comparing different inducers on the same domain, the standard deviation may be seen as an image of the algorithm robustness: if the error on different test sets with the classifiers build on different training sets is very different from one test to another the classification algorithm is not robust to a change of the training set taken from the same dataset.

Now, suppose one wish to compare two algorithms with error rates equals to 9.00 ± 1.00 and 7.50 ± 0.80. Which one is significantly better (at 95% confidence level)? To answer this question, assume the general case to determinate whether the difference between two algorithms — say A_S and A_P — is significant or not. In general, comparisons are made such that A_P is the algorithm proposed and A_S is the standard algorithm. For this, the combined mean and standard deviation are calculated according with Equations 21 and 22, respectively. The absolute difference in standard deviations is computed using Equation 23.

$$\mathsf{mean}(A_S - A_P) = \mathsf{mean}(A_S) - \mathsf{mean}(A_P) \tag{21}$$

$$\mathsf{sd}(A_S - A_P) = \sqrt{\frac{\mathsf{sd}(A_S)^2 + \mathsf{sd}(A_P)^2}{2}} \tag{22}$$

$$\mathsf{ad}(A_S - A_P) = \frac{\mathsf{mean}(A_S - A_P)}{\mathsf{sd}(A_S - A_P)} \tag{23}$$

If $\mathsf{ad}(A_S - A_P) > 0$ then A_P outperforms A_S. Now, if $\mathsf{ad}(A_S - A_P) \geq 2$ standard deviations then A_P outperforms A_S at 95% confidence level. On the other hand, if $\mathsf{ad}(A_S - A_P) \leq 0$ then A_S outperforms A_P and if $\mathsf{ad}(A_S - A_P) \leq -2$ then A_S outperforms A_P at 95% confidence level.

Backing to the same example described earlier, assume $A_S = 9.00 \pm 1.00$, the standard algorithm and $A_P = 7.50 \pm 0.80$, the proposed algorithm. Looking just to the number there is a tendency to say that A_P is better than A_S but using Equations 21, 22 and 23, we obtain:

$$\begin{aligned}
\mathsf{mean}(A_S - A_P) &= 9.00 - 7.50 &&= 1.50 \\
\mathsf{sd}(A_S - A_P) &= \sqrt{\frac{1.00^2 + 0.80^2}{2}} &&= 0.91 \\
\mathsf{ad}(A_S - A_P) &= \frac{1.50}{0.91} &&= 1.65
\end{aligned}$$

Consequently, as $\mathsf{ad}(A_S - A_P) < 2$, A_P **does not** significantly outperform A_S (at 95% confidence level). There are several others statistical tests to measure the significance of any difference between two algorithms, besides the one here described (Freedman et al., 1998). For a good review on comparing predictors, refer to (Salzberg, 1995; Dietterich, 1997b).

2.10 Summary

> It is of interest to note that while some dolphins are reported to have learned English — up to fifty words used in correct context — no human being has been reported to have learned dolphinese.
>
> —*Carl Sagan*

To make machine learning work, we must investigate different structures that may be appropriate for different contexts and understand their strengths and limitations. Machine learning works in many practical applications because the target concepts are not equiprobable and the algorithms are somewhat "aligned" with real world phenomena: features are usually selected by experts and smoothness assumptions that algorithms assume do hold in many situations.

The more we understand about the underlying structures used by classifiers, the more we can modify them based on background knowledge. Similarly, the more we understand the induction algorithms and their assumptions, the easier it is to modify them.

In the next two sections we shall concentrate in general algorithms that learn concepts represented by propositional descriptions such as decision trees — Section 3 — and rule sets — Section 4.

3 Top Down Induction of Decision Trees

> When Kepler found his long-cherished belief did not agree with the most precise observation, he accepted the uncomfortable fact. He preferred the hard truth to his dearest illusions, that is the heart of Science.
>
> —*Carl Sagan*

We start this section explaining basic concepts about inductive learning when acquired knowledge is represented as a decision tree. This sort of algorithms are members of the Top Down Induction of Decision Trees — TDIDT — family.

A Decision Tree — DT — is a recursive data structure defined as:

- a *leaf node* that indicates a class label, or
- a *decision node* that contains a test on a feature value. For each outcome of the test there are one branch and one subtree. Each subtree has the same structure as the tree.

Figure 10 shows an illustrative example of a decision tree for diagnosing a patient. In this figure, each ellipse is a test in one feature from some patient data. Each box is a class label *i.e.*, the diagnosis. To diagnose (classify) a patient we start at the root just following each test down the tree until a leaf is reached.

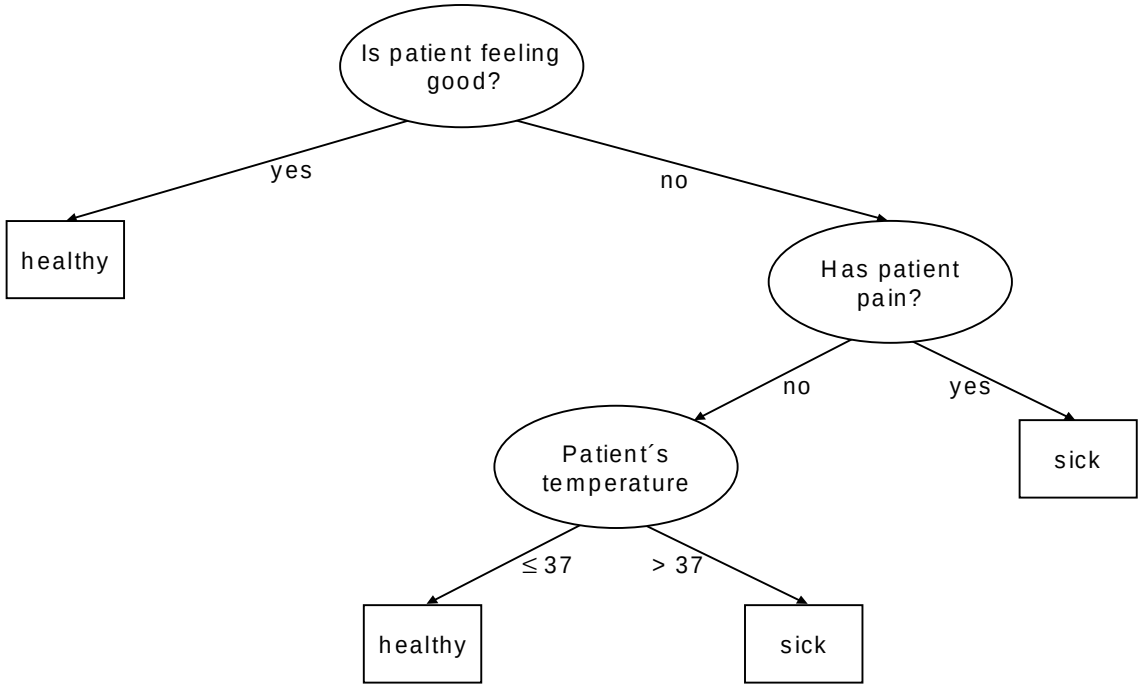


Figure 10: A simple decision tree for diagnosing a patient

It is easy to see that a tree can be represented as a set of rules. Each rule starts at the root and walks down to the leaves. For instance, the tree shown in Figure 10 can be read as:

if is patient feeling good = yes **then**
  class = healthy
else
  if has patient pain = no **then**
    if patient's temperature ≤ 37 **then**
      class = healthy
    else {patient's temperature > 37}
      class = sick
    end if
  else {has patient pain = yes}
    class = sick
  end if
end if

As rules that represent a decision tree are disjunct *i.e.*, only one is fired when a new unlabeled instance is classified, an alternative way of representing these rules would be writing a separate rule for each leaf node, starting at the root, consequently no *else* is really needed:

if is patient feeling good = yes **then**
  class = healthy
end if
if is patient feeling good = no **and** has patient pain = no
and patient's temperature ≤ 37 **then**
  class = healthy
end if
if is patient feeling good = no **and** has patient pain = no
and patient's temperature > 37 **then**
  class = sick
end if
if is patient feeling good = no **and** has patient pain = yes **then**
  class = sick
end if

3.1 Building a Decision Tree

The method for constructing a decision tree from a set T of training instances is surprisingly simple. Let the classes be denoted by $\{C_1, C_2, \ldots, C_k\}$ then the following steps should be followed:

1. T contains one or more instances, all belonging to a single class C_j. In this case, the DT for T is a leaf identifying the class C_j;

2. T contains no instances. Again, in this situation the DT is a leaf but the class associated with the leaf must be determined from information other than T. For example, the most frequent class at the parent of this node could be used.

3. T contains instances that belong to several classes. In this case the idea is to refine T into subsets of instances that are (or seem to be) single-class sets of instances.

   Normally, a test is chosen, based on a single feature, that has one or more mutually exclusive outcomes (in fact, each inducer has its own way to choose the feature to test). Let us denote these outcomes as $\{O_1, O_2, ..., O_r\}$. T is then partitioned into subsets T_1, T_2, ..., T_r, where T_i contains all cases in T that have outcome O_i for the chosen test. The DT for T consists of a internal decision node identified by the chosen test and one branch for each possible outcome.

4. Steps 1., 2. and 3. are applied recursively to each subset of training instances so that the i-th branch leads to the DT constructed from the subset T_i of training instances.

5. After building the DT, pruning may take place (see Section 3.3).

3.2 Choosing the Best Feature to Split

The key to the success of a DT learning algorithm depends on the criterion used to select the feature for splitting. Some possibilities for selecting this feature are:

- Random: select any feature at random;

- Least–Values: choose the feature with the smallest number of possible values;
- Most–Values: choose the feature with the largest number of possible values;
- Max-Gain: choose the feature that has the largest expected information gain *i.e.*, select the feature that will result in the smallest expected size of the subtrees rooted at its children
- Gini (Breiman et al., 1984);
- Gain ratio (Quinlan, 1988).

3.3 Tree Pruning

After building the DT, it is possible that the generated classifier to be very specific for the training data. In this case, we say that the classifier overfits the training data too well. As the training data is only a sample of all possible instances, it is possible to add branches to the tree that improve performance on this training data while decreasing performance on other instances outside this training data. In this situation, the accuracy (or other measure) on an independent (unseen) dataset yields to a poor performance classifier.

Figure 11 — which is similar to Figure 5 — illustrates the impact of overfitting in decision tree learning (Mitchell, 1998). The horizontal axis of this figure indicates the total number of nodes in the DT, as the tree is being constructed. The vertical axis indicates the error of predictions made by the tree. The dotted line shows the error rate of the DT over the training data whereas the solid line shows error rate measured over the test data which is not included in the training data. As expected, the error over the training data (apparent error) decreases monotonically as the tree is grown. However, the error measured over the test data (true error) first decreases up to $N2$ nodes in the tree, then increases.

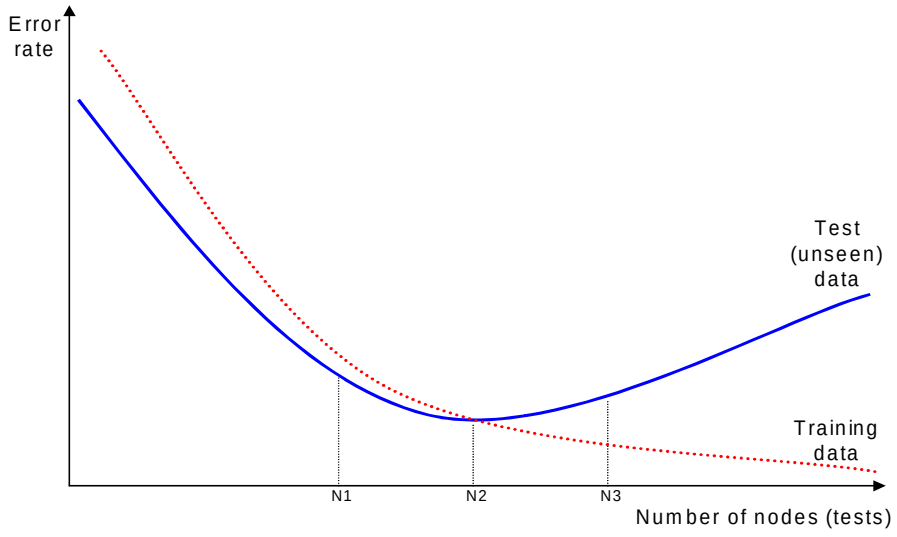


Figure 11: The relationship between tree size and error rate

In order to try not overfitting the data, some inducers *prune* the DT after inducing it. This process, shown in Figure 12, reduces the number of internal test nodes thus reducing the tree complexity while giving a better performance than the original tree. In general, DT inducers

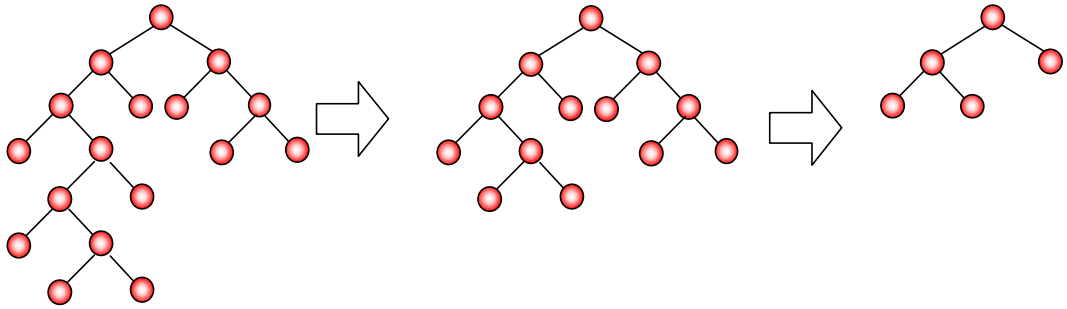


Figure 12: A larger tree is first grown that overfits the data and then pruned back to a smaller (simpler) tree

split by itself the dataset into a training set (which is used for building the DT) and a prune set which is actually used for pruning.

This type of pruning is also called *pos-pruning* since it occurs after building the DT. There are many pos-pruning methods including error-complexity (Breiman et al., 1984) and pessimistic error (Quinlan, 1988).

It is also possible to use *pre-pruning*. This process is done while the DT is induced. However, pre-pruning suffers one side-effect: conjunctions of tests may be the best way for partitioning the instances, but their individual features may not distinguish well the instances. Thus, pre-pruning can prevent the conjunctions from appearing in the tree.

3.4 Classifying New Instances

A DT can be used to classify new instances by starting at the root of the tree and going through each decision node until a leaf is found. When a leaf is encountered, the class label of the new instance is predicted as that one recorded at the leaf.

Remember there is only one path from the root to the leaf of every tree. Each path from the root to each leaf can be considered as a rule. In this case, given an instance it can be classified only by one rule, or in other words, each instance classified by one rule can not be classified by another rule. It is easy to generalize, as already pointed out, that all rules extracted from decision trees are always disjunct.

3.5 Basic Algorithm

The basic TDIDT inducer is illustrated in Algorithm 6.

3.6 An Example

This section shows a running example, adapted from (Quinlan, 1988), for better understanding how to build and use a DT. In order to clarify the difference among the various machine learning approaches considered in this work, the same example will be used in the remaining sections.

Let us suppose a dataset containing day-by-day measures from weather conditions where each instance is composed of the following features:

Algorithm 6 TDIDTs

```
1: procedure TDIDT(Instances)
2: Split Instances into TrainingInstances and PruneInstances
3: tree := generate_tree(TrainingInstances)
4: Pruned_tree := prune(tree,PruneInstances)
5: return Pruned_tree
1: procedure generate_tree(Instances)
2: if termination_condition(Instances) then
3:    Node := majority_class(Instances)
4: else
5:    Best_test := selection_function(Instances)
6:    Node := new_node(Best_test)
7:    for all outcome value O of Best_test do
8:       S := {T in Instances : satisfies(T,Best_test = O)}
9:       Node.subtree[O] := generate_tree(S)
10:   end for
11: end if
12: return Node
```

- outlook: assumes discrete values "sunny", "overcast" or "rain";
- temperature: a numeric value indicating the temperature in Celsius degrees;
- humidity: also a numeric value indicating the percentage of humidity;
- windy: assumes discrete values "yes" or "no" indicating if it is a windy day.

Furthermore, for each day (instance), someone has labeled each day-by-day measure as "go" if weather was nice enough for taking a trip to the farm or "dont_go" if this is not the case. The data could look just like the one shown in Table 9. Although this example has only two possible class labels, it is important to recall that a DT can handle any number k of classes $\{C_1, C_2, \ldots, C_k\}$.

Instance No.	Outlook	Temperature	Humidity	Windy	Voyage?
T_1	sunny	25	72	yes	go
T_2	sunny	28	91	yes	dont_go
T_3	sunny	22	70	no	go
T_4	sunny	23	95	no	dont_go
T_5	sunny	30	85	no	dont_go
T_6	overcast	23	90	yes	go
T_7	overcast	29	78	no	go
T_8	overcast	19	65	yes	dont_go
T_9	overcast	26	75	no	go
T_{10}	overcast	20	87	yes	go
T_{11}	rain	22	95	no	go
T_{12}	rain	19	70	yes	dont_go
T_{13}	rain	23	80	yes	dont_go
T_{14}	rain	25	81	no	go
T_{15}	rain	21	80	no	go

Table 9: The voyage data

Now, let us induce a decision tree from this data. Since the training set T contains instances belonging to more than one class, we need to choose a test based on a single feature. As described in Section 3.2, choosing a feature to split the data depends on each inducer implementation. For this example, let us choose *outlook* as test having three possible outcomes $\{O_1, O_2, O_3\} = \{\text{sunny, overcast, rain}\}$. Then T is partitioned into 3 subsets as shown in Table 10 and its corresponding Figure 13.

Test	Inst.No.	Outlook	Temperature	Humidity	Windy	Voyage?
if outlook = sunny	T_1	sunny	25	72	yes	go
	T_2	sunny	28	91	yes	dont_go
	T_3	sunny	22	70	no	go
	T_4	sunny	23	95	no	dont_go
	T_5	sunny	30	85	no	dont_go
if outlook = overcast	T_6	overcast	23	90	yes	go
	T_7	overcast	29	78	no	go
	T_8	overcast	19	65	yes	dont_go
	T_9	overcast	26	75	no	go
	T_{10}	overcast	20	87	yes	go
if outlook = rain	T_{11}	rain	22	95	no	go
	T_{12}	rain	19	70	yes	dont_go
	T_{13}	rain	23	80	yes	dont_go
	T_{14}	rain	25	81	no	go
	T_{15}	rain	21	80	no	go

Table 10: Building a DT from the voyage data (step 1)

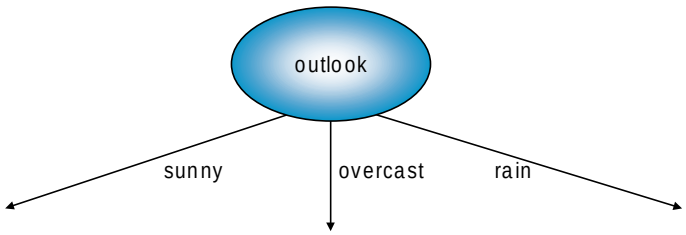


Figure 13: Building a DT from the voyage data (step 1)

As can be seen, each subset still contains instances belonging to several classes, therefore we need to choose a new test based on a single feature. Let us choose *humidity* for the sunny and overcast outcomes and *windy* for the rain outcome. Each subset is now partitioned as shown in Table 11 and its corresponding Figure 14.

After having constructed the complete DT from the data, consider the branch:

if outlook = overcast **then**
  if humidity > 70 **then**
    class = go {Covered instances: T_6, T_7, T_9, T_{10}}
  else {humidity $\leq$ 70}
    class = dont_go {Covered instance: T_8}
  end if
end if

It can be seen that there is only one instance (T_8) for the test "humidity $\leq$ 70"; all other instances for the overcast outcome belong to "class = go". This may indicate an overfitting of the data and the inducer may prune this tree, as can be seen in in Table 12 and its corresponding

Test	Inst.No.	Outlook	Temperature	Humidity	Windy	Voyage?
if outlook = sunny	T_1	sunny	25	72	yes	go
and humidity $\leq$ 78	T_3	sunny	22	70	no	go
if outlook = sunny	T_2	sunny	28	91	yes	dont_go
and humidity $>$ 78	T_4	sunny	23	95	no	dont_go
	T_5	sunny	30	85	no	dont_go
if outlook = overcast	T_6	overcast	23	90	yes	go
humidity $>$ 70	T_7	overcast	29	78	no	go
	T_9	overcast	26	75	no	go
	T_{10}	overcast	20	87	yes	go
if outlook = overcast						
and humidity $\leq$ 70	T_8	overcast	19	65	yes	dont_go
if outlook = rain	T_{11}	rain	22	95	no	go
and windy = no	T_{14}	rain	25	81	no	go
	T_{15}	rain	21	80	no	go
if outlook = rain	T_{12}	rain	19	70	yes	dont_go
and windy = yes	T_{13}	rain	23	80	yes	dont_go

Table 11: Building a DT from the voyage data (step 2)

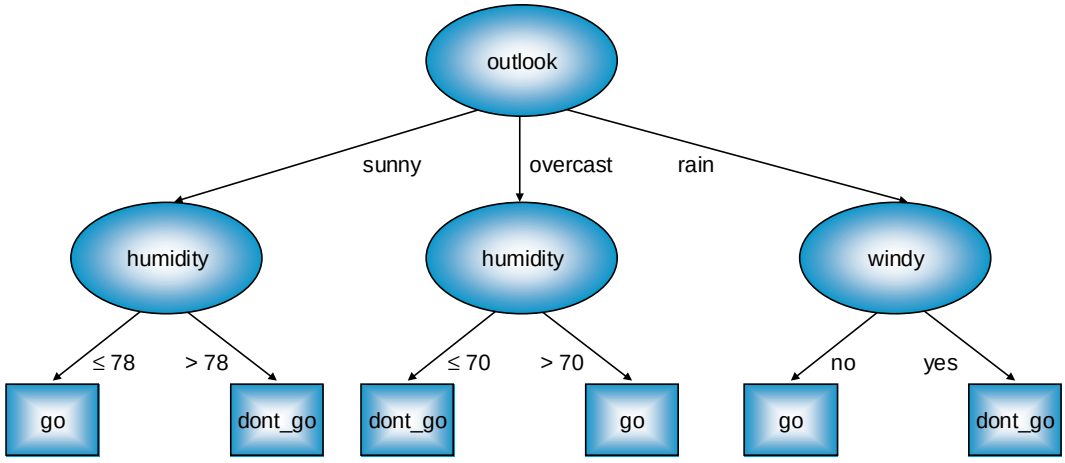


Figure 14: Building a DT from the voyage data (step 2)

Figure 15.

Tree pruning, in general, will improve true classification on unseen data. This may appear counter–intuitive since by pruning some information is thrown away — instance T_8 in this example. However, when learning from noisy data, a correct amount of pruning may improve classification on unseen data. In fact, pruning may eliminates error in data due to noise instead of throwing away relevant information (Bratko, 1990).

3.7 Geometric Interpretation

Considering instances as a vector of m features, such a vector corresponds to a point in the m-dimensional feature space. From this point of view, a DT corresponds to a division (for each test) of this space into regions, each region labeled with a class.

Test	Inst.No.	Outlook	Temperature	Humidity	Windy	Voyage?
if outlook = sunny	T_1	sunny	25	72	yes	go
and humidity $\leq$ 78	T_3	sunny	22	70	no	go
if outlook = sunny	T_2	sunny	28	91	yes	dont_go
and humidity $>$ 78	T_4	sunny	23	95	no	dont_go
	T_5	sunny	30	85	no	dont_go
if outlook = overcast	T_6	overcast	23	90	yes	go
	T_7	overcast	29	78	no	go
	T_8	overcast	19	65	yes	dont_go
	T_9	overcast	26	75	no	go
	T_{10}	overcast	20	87	yes	go
if outlook = rain	T_{11}	rain	22	95	no	go
and windy = no	T_{14}	rain	25	81	no	go
	T_{15}	rain	21	80	no	go
if outlook = rain	T_{12}	rain	19	70	yes	dont_go
and windy = yes	T_{13}	rain	23	80	yes	dont_go

Table 12: Pruning the DT from the voyage data

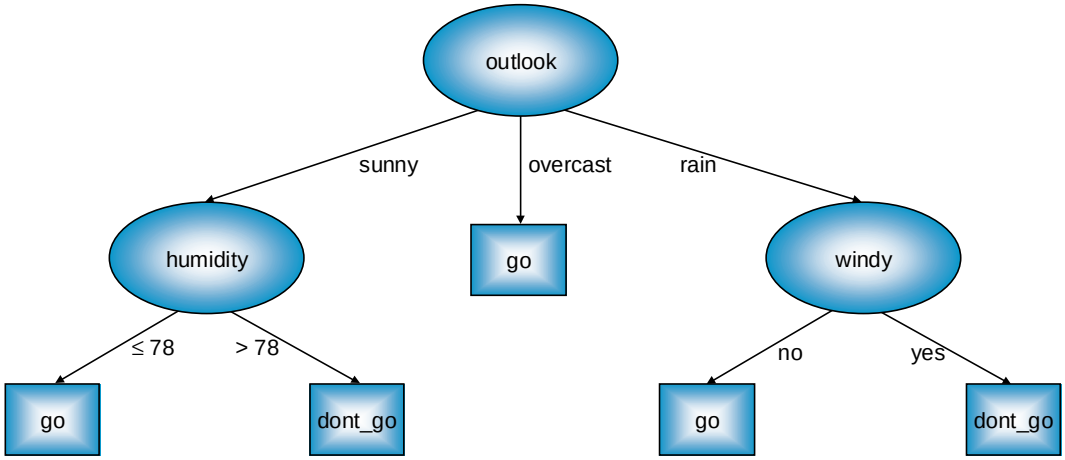


Figure 15: Pruning the DT from the voyage data

3.7.1 Feature-Value

As an example, consider just only two continuous features (X_1 and X_2) and two classes (o and +). Figure 16 shows an example for tests of the following type:

$$X_i \ op \ Value$$

where $X_i \in \{X_1, X_2\}$, *op* is an operator in the set $\{\leq, >\}$ and *Value* is a valid feature X_1 or X_2 constant value.

For this sort of test, the description space is partitioned into rectangular regions, namely hyperplanes that are orthogonal to the axis of the tested feature and parallel to all other features axis. This is an important observation since regions produced by decision trees that use such tests are all hyperrectangles. As the tree is built, more and more regions are added to the space (solid lines).

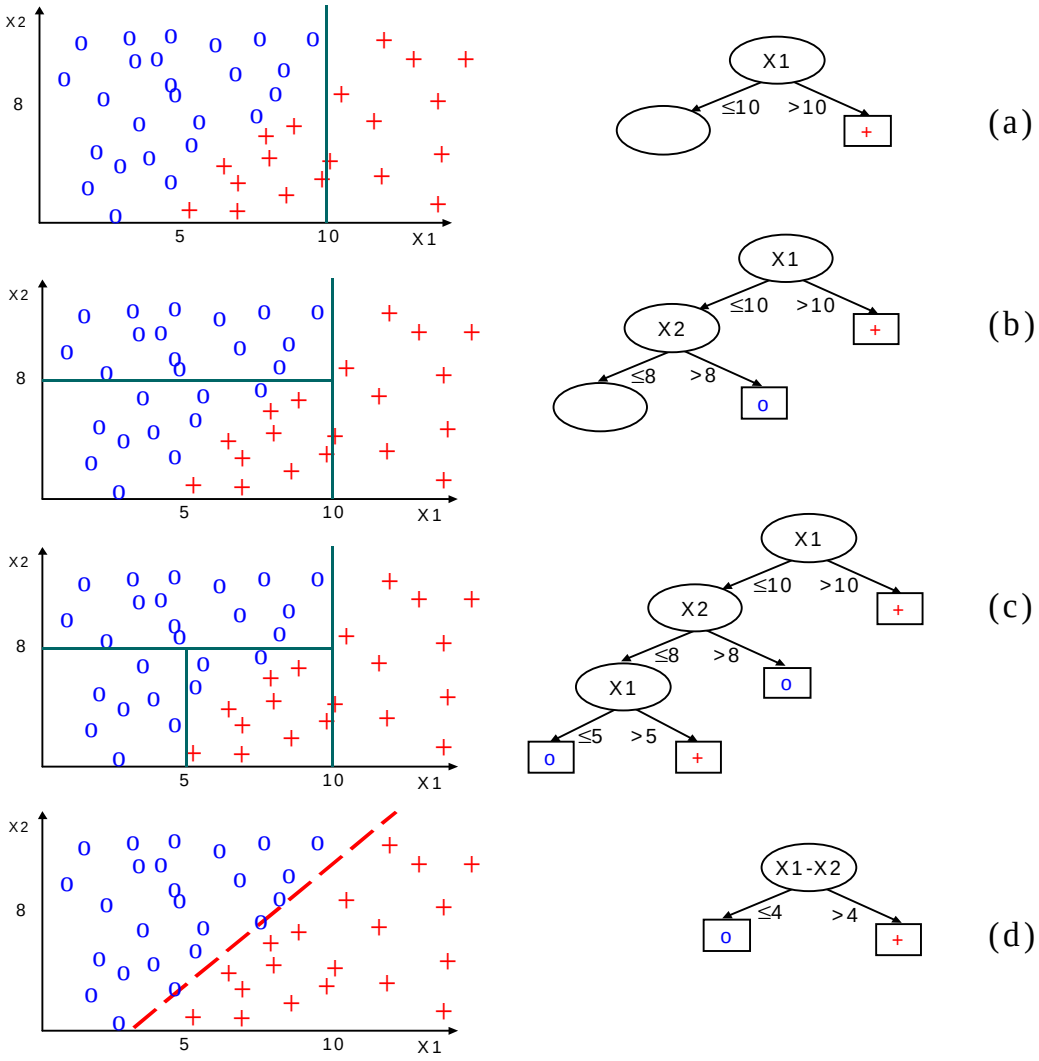


Figure 16: Non-overlapping regions are formed by decision tree in the description space

3.7.2 Linear Combination of Features

Still considering Figure 16, it should be observed that in (d), a simple (dashed line) hypothesis could better classify those instances. This has motivated building *oblique* decision trees which produces non-orthogonal hyperplanes (Breiman et al., 1984, Chapter 5), (Murthy et al., 1994). In this case, tests are of the following type:

$$c_1 \times X_1 + c_2 \times X_2 + \ldots + c_m \times X_m \;\; op \;\; Value$$

where c_i is a constant, X_i is a continuous (integer or real) feature, op is an operator in the set $\{<, \leq, >, \geq\}$ and $Value$ is a constant value. For this sort of test, the description space is partitioned into non rectangular regions, namely hyperplanes that are not necessarily orthogonal to the axis.

3.8 Summary

> The important thing is never to stop questioning.
>
> —*Albert Einstein*

Induction of decision trees is one of the most widely used learning methods in practice. It can outperform human experts in many domains. Some of it strengths include it is a fast method for learning concepts; it is simple to implement; it can convert result to a set of interpretable rules; mature technology, empirically valid in many commercial products; it can handle noisy data.

However there are some weaknesses: larger trees are generally difficult to read, even after tree pruning; it requires fixed–length feature columns; it is non–incremental learning.

Also, univariate (or monothetic) trees which use only one feature at each internal node are limited to axis–parallel partitions of the description space, limiting the concept that may be learned. Oblique (or polythetic) trees which can use more than one feature at each internal node are computationally expensive to induce.

4 Rule Induction

> There are many hypotheses in science which are wrong. That's perfectly all right; they're the aperture to finding out what's right. Science is a self-correcting process. To be accepted, new ideas must survive the most rigorous standards of evidence and scrutiny.
>
> —*Carl Sagan*

As described in the previous section, decision tree induction recursively divides the data into smaller subsets, trying to separate each class from others. Rule induction, on the other hand, induces rules directly. In this process each rule covers a subset of instances that belongs to one specific class.

In general, the Disjunctive Normal Form — DNF — is the most common representation used by many rule inducers. Thus, a rule is a disjunction of conjunctive conditions. Basically, there are two sort of rule induction: ordered and unordered which are explained in the following sections.

4.1 Ordered Rule Induction

The induction of ordered rules works in an iterative way, each iteration searching for a <complex> which covers a large number of instances of a single class C_i and few of other classes $C_j, j \neq i$.

Having found a good <complex>, those covered instances that belong to the class C_i being learned (as well as, eventually, other few instances having class C_j, $j \neq i$ also covered by the same <complex>) are removed from the training set and the rule "if <complex> then class $= C_i$" is added to the end of the rule list. This process iterates until no more complexes can be found.

4.1.1 Basic Algorithm

The basic ordered rule induction (or induction of ordered production rules) inducer is described in Algorithm 7.

Algorithm 7 Ordered rule induction

```
1: procedure induce_ordered_rules(Instances)
2: Rulelist := {}
3: repeat
4:    complex := find_best_conjunction(Instances)
5:    if complex ≠ {} then
6:       C := most_frequent_class(complex,Instances)
7:       Rule := IF complex THEN class = C
8:       Rulelist := Rulelist + {Rule}
9:       Instances := Instances − {T : T in Instances and satisfies(T,complex)}
10:   end if
11: until (Instances = {}) or (complex = {})
12: return Rulelist
```

4.1.2 An Example

The same example used in Section 3.6 will be used here. For simplicity, the dataset in Table 9 is presented again in Table 13.

Instance No.	Outlook	Temperature	Humidity	Windy	Voyage?
T_1	sunny	25	72	yes	go
T_2	sunny	28	91	yes	dont_go
T_3	sunny	22	70	no	go
T_4	sunny	23	95	no	dont_go
T_5	sunny	30	85	no	dont_go
T_6	overcast	23	90	yes	go
T_7	overcast	29	78	no	go
T_8	overcast	19	65	yes	dont_go
T_9	overcast	26	75	no	go
T_{10}	overcast	20	87	yes	go
T_{11}	rain	22	95	no	go
T_{12}	rain	19	70	yes	dont_go
T_{13}	rain	23	80	yes	dont_go
T_{14}	rain	25	81	no	go
T_{15}	rain	21	80	no	go

Table 13: The voyage data

Table 14 shows a rule list, where R_1 is the first induced rule, R_2 is the second one and so on. Also positively covered (PC) instances and negatively covered (NC) instances are shown for each rule. Considering the initial discovered rule R_1, all instances that satisfy rule conditions and are in the same class predicted by the rule conclusion — instances $T_1, T_3, T_7, T_8, T_9, T_{14}, T_{15}$ — as well as those instances that satisfy rule conditions but do not belong to the predicted class go — instances T_{12}, T_{13} — are removed by the ordered rule induction algorithm from the training set and an *else* is introduced in the rule list before inducing the next rule. After that, the next

	Rule	PC	NC
R_1	if humidity < 83 then class = go	$T_1, T_3, T_7, T_8, T_9, T_{14}, T_{15}$	T_{12}, T_{13}
R_2	else if temperature ≥ 23 then class = dont_go	T_2, T_4, T_5	T_6
R_3	else if outlook = rain then class = go	T_{11}	
R_4	else class = dont_go	T_{10}	

Table 14: The voyage ordered rules

rule is induced by the algorithm and the process goes on. Observe that the last rule R_4 is the default rule, which only fires when none of the previous rules R_1, R_2, R_3 fire.

4.1.3 Classifying New Instances

To classify new instances, the ordered rule induction classifier applies an interpretation in which each rule is tried in order until it is found one whose conditions are satisfied by the new instance being classified. The resulting class prediction of this rule is then assigned as the class of that instance. Thus the *order* of rules is important. If no rule is satisfied, the default rule assigns the most common class in the training set to the new instance.

4.1.4 Geometric Interpretation

Ordered rules can be seem as a degenerated binary tree, since to classify an unlabeled instance each rule is tried in order until one fires. As this situation corresponds to an *if-then-elsif* statement, the description space can be considered as partitioned into non-overlapping regions as for decision trees.

4.2 Unordered Rule Induction

The main modification to the ordered rule algorithm is to iterate for each class in turn, but removing only covered instances of that class when a rule has been found. Thus, unlike for ordered rules, the instances from classes C_j, $j \neq i$ incorrectly covered by the current <complex> should remain because now each rule must independently stand against all incorrectly covered instances. Covered instances having the correct class C_i being learned must be removed to stop repeatedly finding the same rule.

4.2.1 Basic Algorithm

The basic unordered rule inducer is described in Algorithm 8.

4.2.2 An Example

The same example used in the previous section will be used here. Table 15 shows a rule set, where R_1 is the first induced rule, R_2 is the second one and so on. Again, positively covered (PC) instances and negatively covered (NC) instances are shown for each rule. Considering the initial discovered rule R_1, instances that satisfy rule conditions and are in the same class predicted by the rule conclusion — instances T_6, T_7, T_8, T_9 — are removed from the training set.

Algorithm 8 Unordered rule induction

```
1: procedure induce_unordered_rules(Instances)
2: Ruleset := {}
3: for class C in Instances do
4:    Rules_for_one_class := {}
5:    repeat
6:       complex := find_best_conjunction(Instances,C)
7:       if complex ≠ {} then
8:          Rules_for_one_class := Rules_for_one_class + {IF complex THEN class = C}
9:          Instances := Instances − {T : T in Instances and satisfies(T,complex and class=C)}
10:      end if
11:   until (Instances = {}) or (complex = {})
12:   Ruleset := Ruleset + Rules_for_one_class
13: end for
14: return Ruleset
```

	Rule	PC	NC
R_1	if outlook = overcast then class = go	T_6, T_7, T_8, T_9	T_{10}
R_2	if outlook = sunny and humidity > 77 then class = dont_go	T_2, T_4, T_5	
R_3	if temperature > 24 then class = go	T_1, T_7, T_9, T_{14}	
R_4	if outlook = rain and windy = no then class = go	T_{11}, T_{14}, T_{15}	
R_5	if outlook = rain and windy = yes then class = dont_go	T_{12}, T_{13}	
R_6	class = go	$T_1, T_3, T_6,$ $T_7, T_9, T_{10},$ T_{11}, T_{14}, T_{15}	$T_2, T_4, T_5,$ T_8, T_{12}, T_{13}

Table 15: The voyage unordered rules

Observe that instances covered by the rule but that do not belong to the same class predicted by rule conclusion — instance T_{10} — remain in the training set. After that, the next rule is induced and the process goes on. As in ordered rules, observe that the last rule R_6 is the default rule, which only fires when none of the previous rules $R_1, R_2, \ldots, R_5$ fire.

4.2.3 Classifying New Instances

To classify a new instance using induced unordered rules, all rules are tried and those which fire are collected. If more than one class is predicted by fired rules, the usual method used is to tag each rule with the distribution of covered instances among classes and then to sum these distributions to find the most probable class. For instance, consider a different example with three induced rules:

if	head=square	**and**	hold=gun	**then**	class=enemy	covers	[15,1]
if	size=tall	**and**	flies=no	**then**	class=friend	covers	[1,10]
if	look=angry			**then**	class=enemy	covers	[20,0]

Here the two classes are {enemy,friend} and [15,1] denotes that the rule covers 15 training instances of enemy and 1 of friend. Given a new instance of a robot which has square head, carries a gun, tall, non-flying and angry, all three rules are fired. Usually, the unordered rule

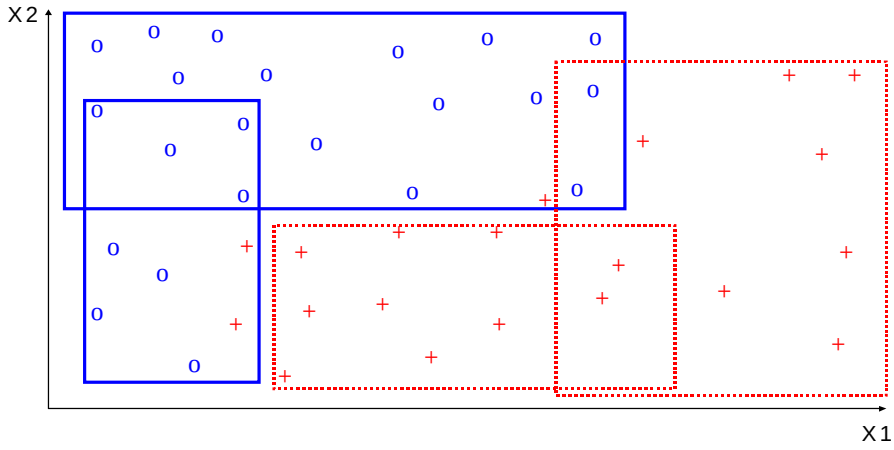


Figure 17: Overlapping regions are generally formed by unordered rule induction in the description space

algorithm resolves this clash by summing the covered instances [36,11] and then predicting the most common class in the sum — enemy.

4.2.4 Geometric Interpretation

Inducing unordered rules divides the description space into overlapping regions, since one instance can be covered by more than one rule.

As an example, consider just only two features (X_1 and X_2) and two classes (o and +). Figure 17 shows four rules (one for each rectangular region): two rectangles with solid lines which represent the two rules having <class = o> and two rectangles with dashed lines representing the two rules having <class = +> . Of course, as in decision trees this division of the description space corresponds to feature-value tests. For linear combination of features, the description space division would look non-orthogonal rectangles.

4.3 Summary

Doubt is the father of invention.

—*Galileo Galilei*

We have presented two basic rule induction algorithms where the order of evaluation of the rules is relevant (ordered rules) and where there is no implicit order among rules (unordered rules).

The strengths of rule induction is comprehensibility and low storage requirements. However, the process of inducing rules is a slower process than inducing decision trees and there are often many variables to tune.

5 Concluding Remarks

I never think of the future. It comes soon enough.

—*Albert Einstein*

In this work we present basic terms commonly used as well as some concepts that are important in the field of Machine Learning, making no assumptions about the reader's background. From the several form of learning we concentrate in inductive supervised learning which is the most researched kind of learning in Artificial Intelligence and has produced many solid results. Specifically, we concentrate in classification, where the class label of the training examples used by the ML algorithm are discrete.

After introducing the basic terminology and concepts, we discuss the importance of empirically evaluating the accuracy of classifiers induced by ML algorithms, and we present some basic statistical methods to estimate classifiers (or hypotheses) accuracy. Also, several methods to combine classifiers (ensembles) in order to improve the accuracy obtained by each individual classifier, are discussed in some detail.

Learning algorithms depend on the language in which datasets and background knowledge are described and the learned concepts are represented. In this work we concentrate in relational descriptions and discuss practical issues in learning decision trees and if-then rules from feature-value datasets.

It should be observed that Machine Learning is a powerful tool for overcoming the bottleneck of knowledge acquisition, and several algorithms have been implemented and also will be implemented in the future. The very basic (but still important) lesson to be learnt is that there is no algorithm that dominates all others for all problems. The best we can hope for is to understand the strengths and limitations of different algorithms, and based on background knowledge for a given domain, make recommendations as to which algorithms to use.

One approach that can work well in practice is to try different algorithms, estimate their accuracy, and pick the one with the highest estimated accuracy for a given domain. This method will work well as long as there are not too many algorithms being tried and when we have reasons to believe that they are appropriate for the domain.

Acknowledgments: We are grateful to Huei Diana Lee and Jaqueline Brigladori Pugliesi for helpful comments on a draft of this report.

References

Agrawal, R., Imielinski, T., and Swami, A. (1993). Mining associations between sets of items in massive databases. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 207–216, Washington D.C.

Agrawal, R. and Srikant, R. (1994). Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Databases.* http://www.almaden.ibm.com/u/ragrawal/pubs.html.

Baranauskas, J. A. and Monard, M. C. (1998a). Experimental feature selection using the wrapper approach. In *Proceedings of the International Conference on Data Mining*, pages 161–170, Rio de Janeiro, RJ. http://www.fmrp.usp.br/~augusto/ps/ICDM98.web.ps.zip.

Baranauskas, J. A. and Monard, M. C. (1998b). Metodologias para seleção de atributos. *Workshop de Teses e Dissertações do Simpósio Brasileiro de Inteligência Artificial (SBIA)*. http://www.fmrp.usp.br/~augusto/ps/SBIA98.web.ps.zip.

Baranauskas, J. A. and Monard, M. C. (1999). The $\mathcal{MLC}$++ wrapper for feature subset selection using decision tree, production rule, instance based and statistical inducers: Some experimental results. Technical Report 87, ICMC-USP. ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/Rt_87.ps.zip.

Baranauskas, J. A., Monard, M. C., and Horst, P. S. (1999b). Evaluation of $\mathcal{CN}2$ induced rules using feature selection. *Argentine Symposium on Artificial Intelligence (ASAI/JAIIO/SADIO)*, pages 141–154. http://www.fmrp.usp.br/~augusto/ps/ASAI99.web.ps.zip.

Baranauskas, J. A., Monard, M. C., and Horst, P. S. (1999a). Evaluation of feature selection by wrapping around the $\mathcal{CN}2$ inducer. *Encontro Nacional de Inteligência Artificial (ENIA/SBC)*, pages 315–326. http://www.fmrp.usp.br/~augusto/ps/ENIA99.web.ps.zip.

Batista, G. E. A. P. A. (1997). Um ambiente de avaliação de algoritmos de aprendizado de máquina utilizando exemplos. Dissertação de Mestrado, ICMC-USP.

Batista, G. E. A. P. A., Carvalho, A. C. P. L., and Monard, M. C. (2000). Applying one-sided selection to unbalanced datasets. In *Proceedings of the Mexican Congress on Artificial Intelligence (MICAI), Lecture Notes in Artificial Intelligence*. Springer-Verlag. (in print).

Batista, G. E. A. P. A. and Monard, M. C. (1998). Descrição da implementação dos métodos estatísticos e de resampling do ambiente AMPSAM. Technical Report 68, ICMC-USP.

Bauer, E. and Kohavi, R. (1999). An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine Learning*, 36:105–142.

Bayardo, R. J. and Agrawal, R. (1999). Mining the most interesting rules. In *Proceedings of the 5th International Conference on Knowledge Discovery and Data Mining*. http://www.almaden.ibm.com/u/ragrawal/pubs.html.

Blake, C., Keogh, E., and Merz, C. J. (1998). UCI repository of machine learning databases. http://www.ics.uci.edu/~mlearn/MLRepository.html.

Blum, A. L. and Langley, P. (1997). Selection of relevant features and examples in machine learning. *Artificial Intelligence*, pages 245–271.

Bratko, I. (1990). *Prolog Programming for Artificial Intelligence*. Addison–Wesley.

Breiman, L. (1996a). Arcing classifiers. Technical report, Statistics Department, University of California, ftp://ftp.stat.berkeley.edu/pub/users/breiman/.

Breiman, L. (1996b). Bagging predictors. *Machine Learning*, 24(2):123–140.

Breiman, L. (1996c). Bias, variance and arcing classifiers. Technical Report 460, Statistics Department, University of California.

Breiman, L. (1996d). The heuristics on instability in model selection. Technical report, Statistics Department, University of California, ftp://ftp.stat.berkeley.edu/pub/users/breiman/.

Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1984). *Classification and Regression Trees*. Wadsworth & Books, Pacific Grove, CA.

Caulkins, C. W. (1999). Aquisição de conhecimento utilizando aprendizado de máquina relacional. Exame de Qualificação de Mestrado, ICMC-USP.

Clark, P. and Boswell, R. (1991). Rule induction with $\mathcal{CN}2$: Some recent improvements. In Kodratoff, Y., editor, *Proceedings of the 5th European Conference (EWSL 91)*, pages 151–163. Springer-Verlag.

Clark, P. and Niblett, T. (1987). Induction in noise domains. In Bratko, I. and Lavrač, N., editors, *Proceedings of the Second European Working Session on Learning*, pages 11–30, Wilmslow, UK. Sigma.

Clark, P. and Niblett, T. (1989). The $\mathcal{CN}2$ induction algorithm. *Machine Learning*, 3(4):261–283.

Cohen, W. W. (1995). Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 115–123, San Francisco, CA. Morgan Kaufmann.

Dietterich, T. G. (1997a). Machine learning research: Four current directions. ftp://ftp.cs.orst.edu/pub/tgd/papers.

Dietterich, T. G. (1997b). Statistical tests for comparing supervised classification learning algorithms. ftp://ftp.cs.orst.edu/pub/tgd/papers.

Efron, B. and Tibshirani, R. (1993). *An Introduction to the Bootstrap*. Chapman & Hall.

Felix, L. C. M., Rezende, S. O., Doi, C. Y., de Paula, M. F., and Romanato, M. J. (1998). $\mathcal{MLC}$++ biblioteca de aprendizado de máquina em $\mathcal{C}$++. Technical Report 72, ICMC-USP. ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/Rt_72.ps.zip.

Flach, P. (1994). *Simply Logical: Intelligent Reasoning by Examples*. John Wiley & Sons Ltd.

Freedman, D., Pisani, R., and Purves, R., editors (1998). *Statistics, Third edition*. W. W. Norton & Company, third edition edition.

Freitas, A. A. (1998a). A multi-criteria approach for the evaluation of rule interestingness. In *Proceedings of the International Conference on Data Mining*, pages 7–20, Rio de Janeiro, RJ.

Freitas, A. A. (1998b). On objective measures of rule surprisingness. In *Principles of Data Mining & Knowledge Discovery: Proceedings of the Second European Symp. Lecture Notes in Artificial Intelligence*, volume 1510, pages 1–9.

Freund, Y. and Schapire, R. E. (1995). A decision-theoretic generalization of on-line learning and an application to boosting. In *Proceedings of the Second European Conference on Computational Learning Theory*, pages 23–37. Springer-Verlag.

Freund, Y. and Schapire, R. E. (1996). Experiments with a new boosting algorithm. In *Proceedings of the Thirteenth International Conference on Machine Learning*, pages 123–140, Lake Tahoe, California. Morgan Kaufmann.

Horst, P. S. (1999). Avaliação do conhecimento adquirido por algoritmos de aprendizado de máquina utilizando exemplos. Dissertação de Mestrado, ICMC-USP.

John, G., Kohavi, R., and Pfleger, K. (1994). Irrelevant features and the subset selection problem. In Kaufmann, M., editor, *Proceedings of the Tenth International Conference on Machine Learning*, pages 167–173, San Francisco, CA.

KDD (1995). *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD-95)*, Menlo Park, CA. American Association for Artificial Intelligence.

Kearns, M. J. and Vazirani, U. V., editors (1994). *An Introduction to Computational Learning Theory*. Ellis Horwood.

Kira, K. and Rendell, L. A. (1992). The feature selection problem: Traditional methods and a new algorithm. In *Tenth National Conference on Artificial Intelligence*, pages 129–134. MIT Press.

Klemettinen, M., Mannila, H., Ronkainen, P., Toivonen, H., and Verkamo, A. I. (1994). Finding interesting rules from large sets of discovered association rules. *Third Internationa Conference on Information and Knowledge Managmente*, 30:401–407.

Kohavi, R. (1997). Wrappers for feature subset selection. *Artificial Intelligence*, 97:273–324.

Kohavi, R. and Sommerfield, D. (1995). Feature subset selection using the wrapper model: Overfitting and dynamic search space topology. In (KDD, 1995), pages 192–197.

Kohavi, R., Sommerfield, D., and Dougherty, J. (1994). *$\mathcal{MLC}$++: A Machine Learning Library in C++*. IEEE Computer Society Press.

Kohavi, R., Sommerfield, D., and Dougherty, J. (1996). Data mining using $\mathcal{MLC}$++: A machine learning library in $\mathcal{C}$++. *Tools with Artificial Intelligence*, pages 234–245.

Kubat, M., Bratko, I., and Michalski, R. S. (1998a). *A Review of Machine Learning Methods*, pages 3–69. In (Michalski et al., 1998).

Kubat, M., Holte, R. C., and Matwin, S. (1997). Learning when negative examples abound: One-sided selection. In *Proceedings of the European Conference on Machine Learning (ECML)*, pages 146–153. Springer-Verlag.

Kubat, M., Holte, R. C., and Matwin, S. (1998b). Machine learning for detection of oil spills in satellite radar images. *Machine Learning*, 30:195–215.

Kubat, M. and Matwin, S. (1997). Addressing the curse of imbalanced training sets: One-sided sampling. In *Proceedings of the Fourteenth International Conference on Machine Learning*, pages 179–186, San Francisco. Morgan Kaufmann.

Langley, P. (1996). *Elements of Machine Learning*. Morgan Kaufmann Publishers, Inc.

Lee, H. D. (2000). Seleção e construção de features relevantes para o aprendizado de máquina. Dissertação de Mestrado, ICMC-USP.

Lee, H. D., Monard, M. C., and Baranauskas, J. A. (1999). Empirical comparison of wrapper and filter approaches for feature subset selection. Technical Report 94, ICMC-USP. ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/Rt_94.ps.zip.

Margineantu, D. D. and Dietterich, T. G. (1997). Pruning adaptive boosting. In *Proceedings of the Fourteenth International Conference on Machine Learning*, pages 211–218, San Francisco. Morgan Kaufmann.

Michalski, R. S. (1983). A theory and methodology of inductive learning. *Artificial Intelligence*, 20:111–161.

Michalski, R. S., Bratko, I., and Kubat, M., editors (1998). *Machine Learning and Data Mining Methods and Applications*. John Wiley & Sons Ltd., West Sussex, England.

Michalski, R. S. and Kaufman, K. A. (1998). *Data Mining and Knowledge Discovery: A Review of Issues and a Multistrategy Approach*, pages 71–112. In (Michalski et al., 1998).

Michie, D. (1988). Machine learning in the next five year. In *Proceedings of the Third European Working Session on Learning EWSL-88*, Glasgow, London, Pitman.

Michie, D., Spiegelhalter, D. J., and Taylor, C. C., editors (1994). *Machine Learning, Neural and Statistical Classification*. Ellis Horwood.

Mitchell, T. M. (1998). *Machine Learning*. McGraw–Hill.

Morik, K., Wrobel, S., J *Knowledge Acquisition and Machine Learning: Theory, Methods, and Applications*. Harcourt Brace & Company, Publishers.

Moses, L. E., editor (1986). *Think and Explain with Statistics*. Addison–Wesley.

Muggleton, S. H. and Raedt, L. D. (1994). Inductive logic programming. *The Journal of Logic Programming*, 19-20:629–679.

Murthy, S. K., Kasif, S., and Salzberg, S. L. (1994). A system for induction of oblique decision trees. *Journal of Artificial Intelligence Research*, 2(1):1–32. http://www.cs.jhu.edu/~salzberg/jair94.ps.

Prati, R. C., Baranauskas, J. A., and Monard, M. C. (1999). BibView: Um sistema para auxiliar a manutenção de registros para o BibTeX. Technical Report 95, ICMC-USP. ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/Rt_95.ps.zip.

Quinlan, J. R. (1988). *C4.5 Programs for Machine Learning*. Morgan Kaufmann, CA.

Quinlan, J. R. (1990). Learning logical definition from relations. *Machine Learning*, 5:239–266.

Quinlan, J. R. (1996). Bagging, boosting and $\mathcal{C}4.5$. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 725–730. American Association for Artificial Intelligence.

Rathjens, D. (1996). *MineSet™ User's Guide*. Silicon Graphics, Inc.

Russel, S. and Norvig, P. (1994). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Salzberg, S. L. (1995). On comparing classifiers: A critique of current research and methods. Technical Report JHU-95/06, Department of Computer Science, Johns Hopkins University. http://www.cs.jhu.edu/~salzberg/critique.ps.

Schapire, R. E. (1990). The strength of weak learnability. *Machine Learning*, 5(2):197–227.

Simon, H. (1983). Why should machines learn? In Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., editors, *Machine learning: An Artificial Intelligence Approach*, volume 1. Morgan Kaufmann.

Sterling, L. and Shapiro, E. (1994). *The Art of Prolog, 2nd edition*. The MIT Press.

Weiss, S. M. and Indurkhya, N. (1998). *Predictive Data Mining: A Practical Guide*. Morgan Kaufmann, San Francisco, CA.

Weiss, S. M. and Kulikowski, C. A. (1991). *Computer Systems that Learn*. Morgan Kaufmann, San Mateo, CA.

Wolpert, D. H. (1992). Stacked generalization. *Neural Networks*, 5:241–259.